Technical Report ITL-95-9
September 1995

US Army Corps of Engineers
Waterways Experiment Station

A Study of the Rational Environment

by William A. Ward, Jr.,
University of South Alabama

Prepared for U.S. Army Environmental Center

PRINTED ON RECYCLED PAPER

Technical Report ITL-95-9
September 1995

A Study of the Rational Environment

by William A. Ward, Jr.

Faculty Court West 20
School of Computer and Information Sciences
University of South Alabama
Mobile, AL 36688

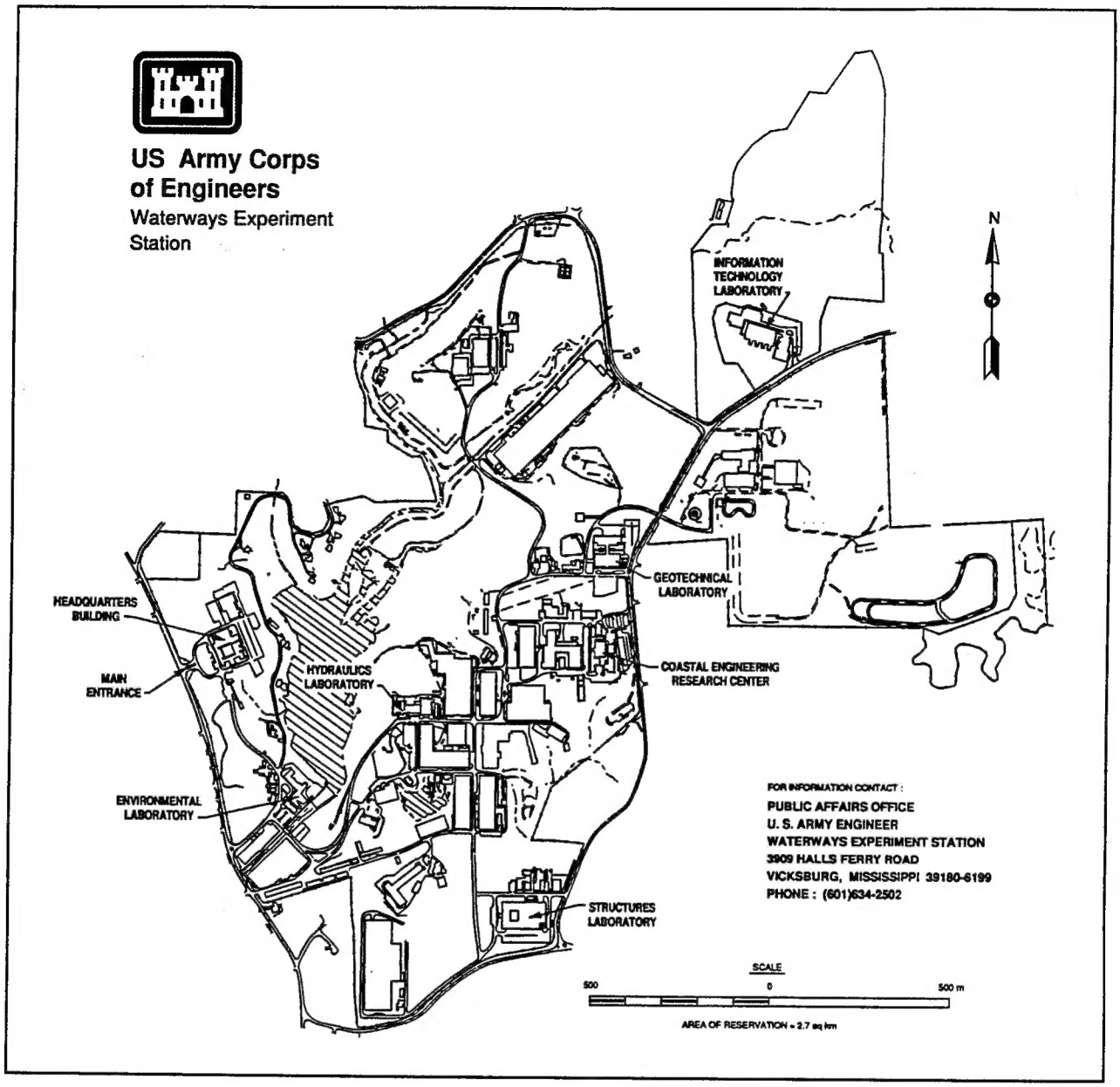
Final report

Approved for public release; distribution is unlimited

Prepared for U.S. Army Environmental Center
Building E4435, Edgewood Area
Aberdeen Proving Ground, MS 21010

Under Contract No. DACA39-93-K-0016

Monitored by U.S. Army Engineer Waterways Experiment Station
3909 Halls Ferry Road, Vicksburg, MS 39180-6199



Waterways Experiment Station Cataloging-in-Publication Data

Ward, William A.

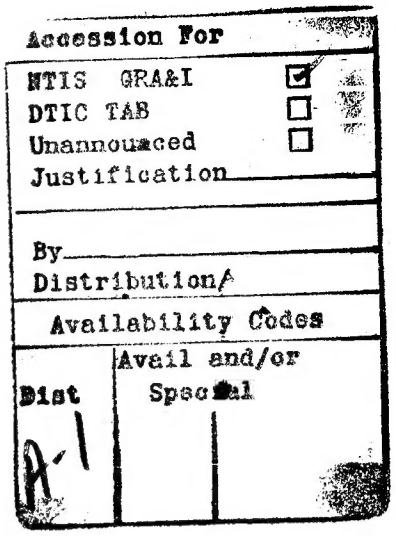
A study of the rational environment / by William A. Ward, Jr. ; prepared for U.S. Army Environmental Center ; monitored by U.S. Army Engineer Waterways Experiment Station.

70 p. : ill. ; 28 cm. -- (Technical report ; ITL-95-9)

Includes bibliographic references.

1. Ada (Computer program language) 2. Software engineering. I. United States. Army. Corps of Engineers. II. U.S. Army Engineer Waterways Experiment Station. III. Information Technology Laboratory (U.S. Army Engineer Waterways Experiment Station) IV. U.S. Army Environmental Center. V. Title. VI. Series: Technical report (U.S. Army Engineer Waterways Experiment Station) ; ITL-95-9

TA7 W34 no.ITL-95-9



Contents

List of Figures

List of Tables

Preface

This report is published in the interest of scientific and technical information exchange; the ideas and findings contained herein should not be construed as an official position of the U.S. Army Corps of Engineers. Use of any trademarks in this report is not intended in any way to infringe on the rights of the trademark holder.

The author thanks Dr. Windell F. Ingram for reviewing this report, and Mr. Jeffrey S. Martell and Mr. Sean S. Kae of Rational for supplying much useful information on their company's products, as well as providing the author the opportunity for hands-on use of the Rational Environment. Thanks are also due to the Rational users who took time to discuss their experiences with the Rational Environment; Ms. Hilary Allers of TRW Federal Systems Group, Ms. Jennie Beckley of IBM, Mr. David Kriegman of SRA, and Mr. Bob Schoenborn of Statistica.

The production of this report was sponsored by the U.S. Army Environmental Center (AEC) and funded through the U.S. Army Engineer Waterways Experiment Station (WES) Information Technology Laboratory (ITL) under Contract No. DACW39-93-K-0016 from March 3, 1993 to December 31, 1993.

Mr. Mark N. Bovelsky was Chief of the Information Management Branch, AEC, during the preparation of this report. The contract was monitored by Dr. Windell F. Ingram, Chief, Computer Sciences Division, ITL. Dr. N. Radhakrishnan was Director, ITL, Dr. Robert W. Whalin was Director of WES, and COL Bruce K. Howard, EN, was Commander.

1 Introduction

This report is intended to be a comprehensive survey of publicly available information on the Rational Environment[1]. Its primary purpose is to introduce potential users of the system to its capabilities by describing its current features and summarizing user experiences. As such, it will also be of interest to students and researchers in the area of software development environments. It must be emphasized that no formal evaluation of the Environment was conducted as a part of this study and that it is not the intent of this report to suggest a methodology for evaluating this, or other, environments. Readers wishing to do so should seek guidance from other sources (Lyons and Nissen 1990, Firth et al. 1987, Lyons and Nissen 1986, Weiderman et al. 1986, Wood et al. 1988).

The report begins by presenting the historical context in which the Environment was developed. So that its use may be more clearly understood by those who have never used software development aids of this type, a brief, general discussion of programming and project support environments is given as background. This is followed by a description of the Environment itself and tools which may be added to enhance it. The experiences of several Rational users are reported, along with the results of some formal evaluations of the product. A final section presents recommendations on how to successfully use the Environment and some conclusions regarding its capabilities.

Historical Perspective

Understanding the Environment is difficult without understanding the three decades of software development that preceded its creation. The first software systems were short and simple. These so-called systems hardly qualified as such, generally being standalone utilities a few hundred source lines of code (SLOC) long designed to sort a list, solve a linear system, or the like; furthermore, they required little manpower to write and maintain, often being created by a single programmer.

[1] For the sake of brevity, it will often be referred to as "the Environment."

Since that time software projects have increased in scope, attempting to perform more complex tasks, growing to hundreds of thousands or millions of SLOC, and requiring teams of dozens, if not hundreds, of programmers. Examples include operating systems, telephone switching systems, and embedded weapons systems.

Aside from their sheer size, the complexity of these development projects is increased by several other factors. Often the software must run on multiprocessors, or on communicating nodes in a network of computers; in such situations, deadlock avoidance and asynchronous event handling are often necessary. These large systems typically have long lifetimes and frequently thus involve extensive error correction and/or addition of features. Finally, some are intended from the beginning to run on a variety of platforms, or migrate to other hardware in mid-life. As might be expected, many such large-scale software development projects have been less than completely successful; Brooks opens *The Mythical Man Month* with an eloquent description of the situation:

> No scene from prehistory is quite so vivid as that of the mortal struggles of great beasts in the tar pits. In the mind's eye one sees dinosaurs, mammoths, and saber-toothed tigers struggling against the grip of the tar. The fiercer the struggle, the more entangling the tar, and no beast is so strong or so skillful but that he ultimately sinks.
>
> Large-system programming has over the past decade been such a tar pit, and many great and powerful beasts have thrashed in it. Most have emerged with running systems—few have met goals, schedules, and budgets. Large and small, massive or wiry, team after team has become entangled in the tar. No one thing seems to cause the difficulty—any particular paw can be pulled away. But the accumulation of simultaneous and interacting factors brings slower and slower motion. Everyone seems to have been surprised by the stickiness of the problem, and it is hard to discern the nature of it. But we must try to understand it if we are to solve it.[1]

In spite of the understandable reluctance of Government contractors to disclose the degree to which their systems fail to meet specifications, there is evidence that Government software projects are particularly susceptible to these problems. One study evaluated the success of nine software development projects performed for the Department of Defense (DoD) (Comptroller General of the United States 1979). The value of these contracts was $6.8 million; that total expenditure may be broken down as follows: $3.2 million for software that was delivered to the Government but never successfully used, $1.95 million for software that was never delivered, $1.3 million for poor quality software requiring extensive modification or ultimate abandonment, $198,000 for software that had to be modified before being used, and only $119,000 for software used as

[1] From (Brooks 1975, p. 4); © 1975 Addison-Wesley, reprinted by permission.

delivered. Although small in scope, this study unfortunately seems to be representative of software development in general. (Other more recent fiascos are noted in (Brown, Earl, and McDermid 1992, p. 8-9))

What are the reasons for this sorry state of affairs? Before answering this question, it will be useful to separate causes from effects. Of the latter, seven stand out (Fisher 1976, p. 2-3). Software does not satisfy user's needs; it is not *responsive*. Software fails; it is *unreliable*. Software costs to much and the costs are unpredictable; it is *expensive*. Software is difficult to modify; it is *unmaintainable*. Software is delivered late and without all the specified features; it is not *timely*. Software is difficult to move from one system to another; it is not *portable*. Finally, software consumes too much processor time and memory; it is *inefficient*.

As noted in (Booch 1987, p. 7-10) and (Devlin 1980, p. 2), however, these are symptoms of other more fundamental problems, five of which are described below. (1) Organizations responsible for supervision of software development, as well as those involved in the development process itself, do not understand the implications of good software engineering practice (or the lack thereof) on a project. (2) There is a continuing shortage of trained software engineers. Indeed, many computer science programs do not even offer courses in software engineering. When they do, the courses are often optional and viewed as not central to the discipline (Frakes, Fox, and Nejmeh 1991, p. 3). (Refer to (Gibbs and Ford 1986) and (Shaw 1986) for further information on this issue.) (3) Von Neumann architectures discourage good software engineering practice.

> Conventional programming languages are growing ever more enormous, but not stronger. Inherent defects at the most basic level cause them to be both fat and weak: their primitive word-at-a-time style of programming inherited from their common ancestor—the von Neumann computer, their close coupling of semantics to state transitions, their division of programming into a world of expressions and a world of statements, their inability to effectively use powerful combining forms for building new programs from existing ones, and their lack of useful mathematical properties for reasoning about programs.[1]

(4) Programmers view their activity as an art and resist using new methodologies that would improve their efficiency. Ironically, programming teams that insist on replacing last year's system with the latest hardware are quite content to use development techniques thirty years old. (5) Organizations consistently underestimate the difficulties associated with and the resources required for the development of large software systems. They add personnel to large projects with a misguided optimism similar to the belief that if one woman can produce one baby in nine months, then nine women can produce one baby in one month. Furthermore, project leaders who would agree that it is impossible to build a 100-story

[1] From (Backus 1978, p. 613); © 1978 ACM, reprinted by permission.

skyscraper with the same tools and materials used for a two-story townhouse, nevertheless ask their developers to create million SLOC applications with the same tools they used for those of 1000 SLOC.

Further discussion of issues related to the creation of large software systems may be found in (Newport 1986) and (Wilson 1987). For a more thorough exploration of the issues related to large-scale software system development see (Brooks 1975, Byrne 1991, Feiler and Smeaton 1988).

Program Development Environments

Just as programming languages have evolved, so have tools for software development. Originally developers were equiped with no more than assemblers, compilers, linkers, and loaders. Interactive time-sharing systems made possible interactive line editors, and then full-screen editors, as well as other tools. By 1980 a variety of tools were available on time-sharing systems to support the implementation phase of the software life cycle (Barstow, Shrobe, and Sandewall 1984, Hünke 1981). That same year, of course, also marked the introduction of Ada as the DoD language of choice for embedded systems. The development tools then available for these systems were still minimal, typically including only a compiler, linker, and editor, as well as being poorly integrated.

The DoD recognized that instituting a standard language was only the first step in addressing the software crisis; an Ada programming support environment (APSE) to facilitate development of programs in Ada would also be necessary. Using discussions from a 1978 workshop (Standish 1978) as initial input, DoD followed the Ada model by issuing a series of increasingly more specific APSE requirement documents: SANDMAN, PEBBLEMAN (High Order Language Working Group 1978, High Order Language Working Group 1979), and STONEMAN (High Order Language Working Group 1980, Buxton and Druffel 1980). The STONEMAN APSE, as originally envisioned by DoD, would span the entire software life cycle. Furthermore, it was to be an integrated environment; APSE tools would not only access the Ada program under development, but would also be capable of communicating with other tools in the environment. Finally, the APSE was to have an intuitive, easy-to-use interface.

The STONEMAN model, shown in Figure 1, consists of four layers. Level 0, the host hardware and software, is the lowest of these and serves as the foundation for the other layers.

Level 1, called the *kernel APSE* (KAPSE), contains all the facilities necessary to execute Ada programs. A central feature of the KAPSE is an object database containing, for example, the APSE tools themselves, other Ada program units, test data, and program designs. The KAPSE also includes an Ada run-time system and mechanisms for tool intercommunication and database access.

Level 2 is the *minimal APSE* (MAPSE); it provides those functions which were deemed both necessary and sufficient for Ada program development and

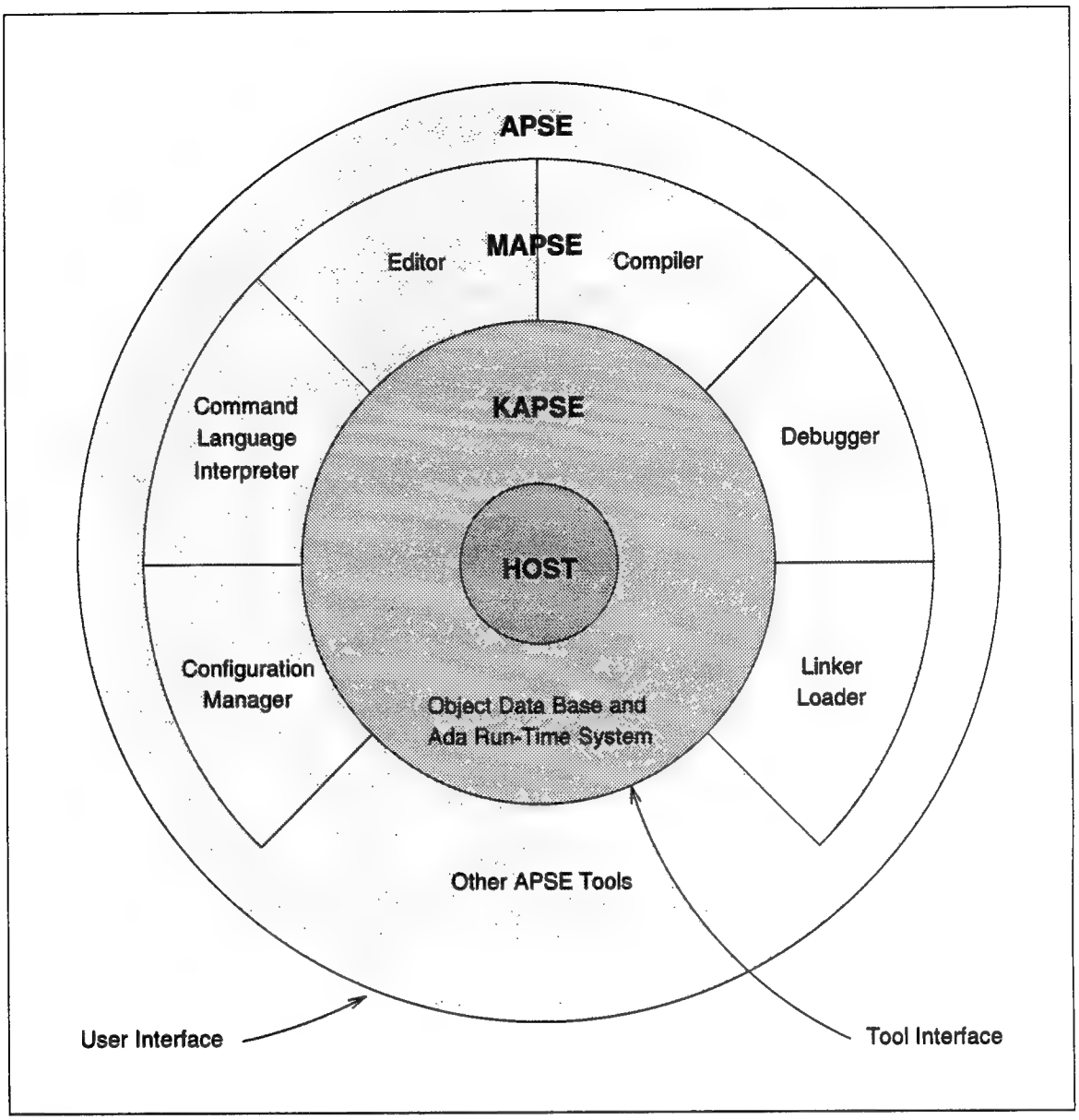


Figure 1. The STONEMAN APSE model (adapted from (Hitchon et al. 1989, p. 4))

maintenance. Twelve tools were included in this basic set: compiler, text editor, pretty printer, linker, loader, variable cross referencer, static analyzer, dynamic analyzer, terminal interface, file manager, simple command interpreter, and configuration manager (Sommerville and Morrison 1987).

Level 3 is the *full APSE*. It extends the MAPSE by providing additional support for specific applications or design methodologies. Examples of such extensions include an Ada-sensitive editor, documentation system, configuration management and version control (as distinct from just configuration management), fault reporting system, project management tool, software metrics

collectors, support for requirements analysis and system design, automatic program verifiers, and an Ada-based command interpreter (Sommerville and Morrison 1987).

Since the release of STONEMAN, computer-aided software engineering (CASE) has grown, both as a research area and as an industry. The commercialization of the field has resulted in coverage of all phases of the development process by a vast number of tools; furthermore, techniques for integrating these tools in a coherent fashion has become an important field of study (Brown and Penedo 1992, Morris, Feiler, and Smith 1991, Wasserman 1990).

When an amalgam of tools purports to cover two or more phases, the result is often referred to by its vendor as an "environment." To clarify what is meant by this term and, more specifically, to understand the scope of applicability of the Environment, it will be useful to present the SEE taxonomy of Sommerville and Morrison.[1]

- *Teaching and learning environments* are intended for use by beginning programmers. Their syntax and debugging aids free the novice programmer to concentrate on problem solving and program development. The Cornell Program Synthesizer (Teitelbaum and Reps 1981) is an example of such a system.

- *Nonprofessional environments* are designed to support rapid program development by users who are not professional programmers. UCSD Pascal and the versions of BASIC available on most microcomputers fall into this category.

- *Language-oriented environments* provide professional programmers with an integrated suite of tools for developing software in a particular language. "The programmer using the supported language L sees, in effect, an L-machine rather than a brand X computer running some operating system which includes a compiler for L."[2] Smalltalk environments (Deutsch 1985, Goldberg and Robson 1983, Krasner 1983) are examples of this class.

- *General-purpose environments* are intended for professional programmers and are language independent. Their tool support for implementation and testing is extensive, but that for other phases is typically sparse. The UNIX[3] Programmer's Workbench (Dolotta, Haight, and Mashey 1978, Kernighan and Mashey 1981, Mitze 1989) is the prime example of this category.

---

[1] The category titles used in this taxonomy are from (Sommerville and Morrison 1987, pp. 27-28); © 1987 Addison-Wesley Publishers Limited, reprinted by permission.

[2] From (Sommerville and Morrison 1987, p. 28); © 1987 Addison-Wesley Publishers Limited, reprinted by permission.

[3] UNIX is a trademark of X/Open.

- *Software design environments*, as the name implies, focus on the design phase of the software life cycle. These are typically graphically-oriented tools which support a particular design methodology. The AIDES environment, PRISM, and the Analyst are examples of this class.

- *Integrated project support environments* (IPSEs) are the most sophisticated, including tools for requirements analysis, design, specification, code development, office automation, and project management. Because they provide facilities to support the entire software life cycle, they reach their full potential when applied to the development of large software systems. ISTAR is an example of this type of environment (Graham and Miller 1988, Stenning 1986).

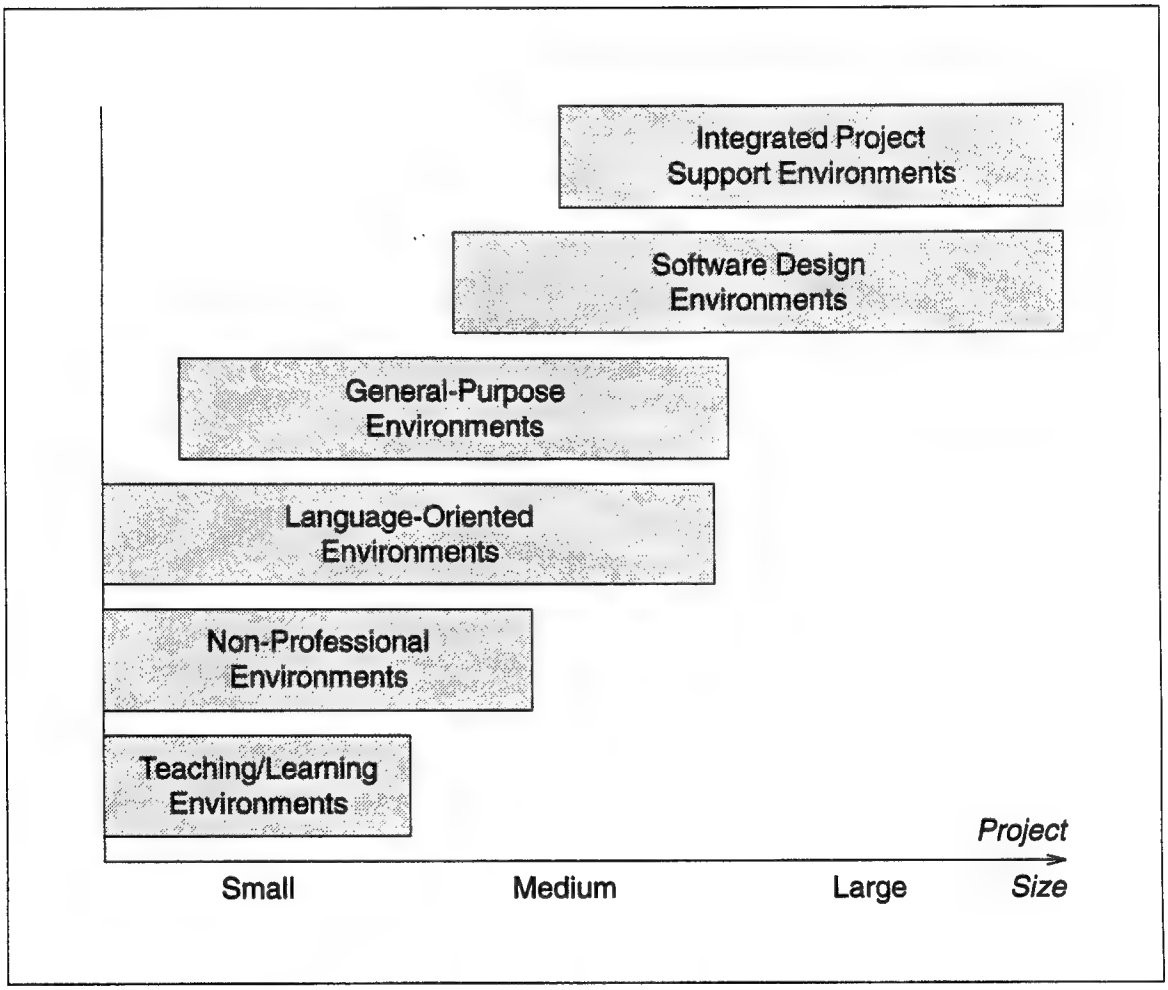


Figure 2. Range of applicability of various types of environments (from (Sommerville and Morrison 1987, p. 29); © 1987 Addison-Wesley Publishers Limited, reprinted by permission)

Using Sommerville's taxonomy, the full APSE is probably best viewed as a language-specific IPSE. The relative range of applicability of these environments is illustrated in Figure 2. Further information on SEEs is available from a number of sources, including (Barstow, Shrobe, and Sandewall 1984, Bennett 1989, Brown, Earl, and McDermid 1992, Hünke 1981, Long 1990, Sommerville 1986). A classification of case technology, including a discussion of the relationship of various types of environments to CASE technology in general has recently been presented in (Fuggetta 1993).

How well do such environments address the various phases of the software life cycle? Unfortunately, as noted in (Schefström 1990), there is a dichotomy in CASE technology. At one end of the spectrum are "back-end" CASE environments, such as Interlisp and Smalltalk-80, while at the other are "front-end" CASE tools such as Cadre's Teamwork, IDE's Software through Pictures (StP), and Rational Rose. The former assist programmers by focusing on implementation and testing, while the latter assist project managers with specification and design. As will be seen, the basic, unadorned Environment is a back-end tool.

Front-end and back-end CASE tools are typically not well integrated; the resulting gap between tools for the "generals" and tools for the "troops" makes it difficult to move from design to implementation. Furthermore, because many software development efforts are preceded by business process modeling or reengineering (D. Appleton Company 1992), and because the software tools which support these activities communicate poorly with front-end CASE tools, there is a second tool integration gap. This overall situation is illustrated in Figure 3.

Where does the Environment fit in this taxonomy, and to what degree does it suffer from the discontinuities noted above? Schefström, in the same paper noted above, stresses the need for a "homogeneous CASE" environment built around a "semantically unified internal form" and containing a "design editor", "program editor", and "(incremental) compiler/checker".[1] He elaborates on the Environment's capabilities in the following complimentary yet critical evaluation.

> The effort that most closely adhered to the original vision of an APSE was however the Rational Environment. Starting as a small venture capital company, they took the spirit from Stoneman, Interlisp, Smalltalk, and the language oriented editor approaches, and implemented a tightly integrated development environment for Ada. Integration, and tailoring for the purpose, was taken very far, with a special purpose hardware supporting an operating system that is completely dedicated towards production of Ada software. The compiler was built to be incremental, and most tools work against the internal representation of the programs.

[1] These phrases are from (Schefström 1990, pp. 133,135); © 1990 Cambridge University Press, reprinted by permission.

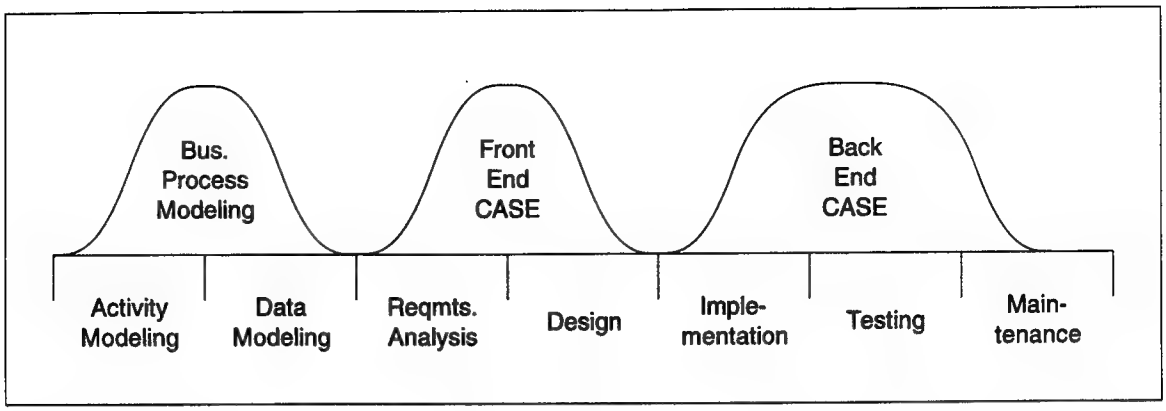


Figure 3. Spectrum of applicability of life cycle tool types (adapted from (Schefström 1990, p. 132); © 1990 Cambridge University Press, reprinted by permission)

> While almost all other language oriented editors provided *syntax* oriented support, whose importance in a broader perspective can be questioned, the Rational environment could provide further services like interactive cross referencing and semantic completion. This, together with a number of well integrated services for configuration control and documentation support, made Rational be an environment that people really liked to work with. The same property that made it initially possible to explore the benefits of tight integration, was at the same time however a problem. The special purpose hardware and proprietary operating system, together with the implied major investment, can make many potential users hesitate. The Rational environment is however one of the few novel environments that has been made a stable product in industrial use.[1]

In spite of this criticism, the Environment has been used successfully on several very large (1,000,000+ SLOC) projects; in that sense it qualifies as an IPSE. Furthermore, as will be noted in a subsequent section, Rational has responded to this criticism with a significant new product. However, perhaps the true test of the effectiveness of the Environment is the degree to which it supports the various phases of the software life cycle, and at the same time closes the tool integration gaps noted above. The remainder of this report will focus on how well the Environment meets that test.

---

[1] From (Schefström 1990, p. 129); © 1990 Cambridge University Press, reprinted by permission.

2 Description of the Rational Environment

Overview

Rational refers to its strategy for software engineering with Ada as "Rational Control." The four parts of this strategy are (1) the Rational Environment itself, (2) the ability to integrate many popular third party CASE tools and development aids into the Environment, (3) interfaces for Ada compilers on almost every platform, and (4) a dedication to customer service. Together, these components address a broad spectrum of software development activities, from analysis and design, through implementation, to testing and maintenance. How this is accomplished will be discussed in the following sections (Rational 1992 Document D-81).

The product which serves as the cornerstone of Rational Control is the Rational Environment. Broadly speaking, the Environment is intended to provide, in a highly integrated fashion, tools and capabilities to address all aspects of the software life cycle. It is intended to facilitate the development of large-scale software systems in Ada. Appropriately, the Environment is itself a large Ada system and as such was developed and is maintained using its own facilities.

As illustrated in Figure 4, the Environment has a layered structure; at the core is the information repository, a database of information on the system under development. The next layer is a programmatic interface to this database which provides a consistent mode of access to the information. Rational's Configuration Management and Version Control (CMVC) uses this interface to maintain program consistency and integrity and to facilitate the activities of teams of developers working on multiple versions of a large program. The outer layer consists of a graphical user interface (GUI) equipped with a suite of tools with which the developer interacts. Examples of these tools include Rational's Ada language-sensitive editor and debugger and third-party CASE tools such as Cadre's Teamwork. Together, the features provided by this layered approach

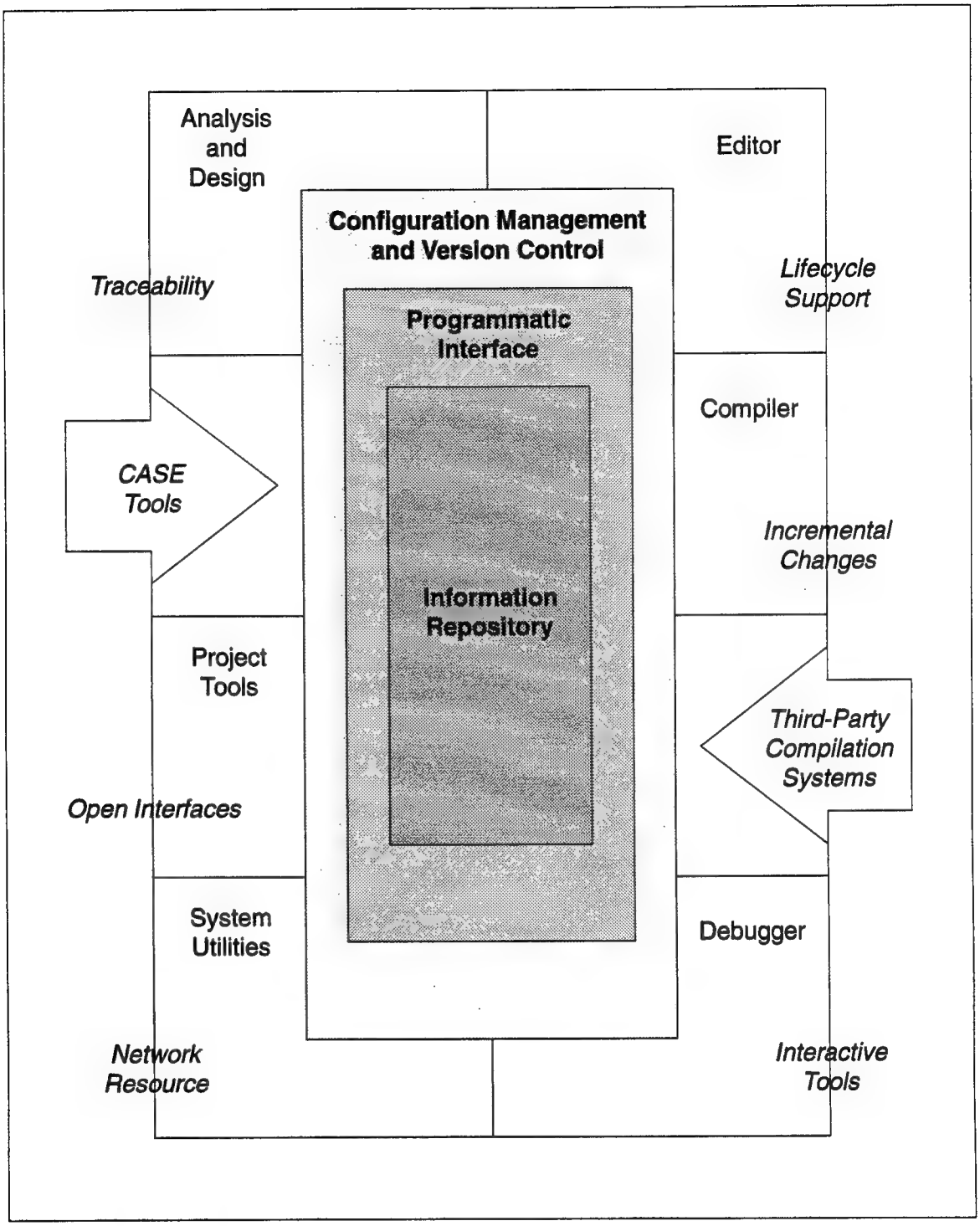


Figure 4. Logical relationship between environment components (from (Rational 1992 Document D-81, p. 5))

provide an effective means to produce Ada software (Rational 1992 Document D-76).

The R1000 Software Engineering Server

The prototype of the Environment was developed during the early 1980s. Testing of this prototype convinced its implementors that the then current hardware platforms were several orders of magnitude too slow to support the type of interactive software development they had envisioned. As a result, Rational built a processor specifically designed to support the Environment. The current incarnation of this system is the R1000 Model 400.

Whereas most system designers focus on the implementation of a relatively low-level instruction set, the architects of the R1000 concentrated on providing hardware features which would speed the development, compilation, and execution of Ada programs. This was a formidable task, since Ada requires strong type checking across module interfaces. Furthermore, changes to statements and data definitions require recompilation of dependent modules to maintain program consistency; this can result in cascading recompilation of numerous packages and modules.

The required speed is obtained through parallelism, wide data paths, and hardware assists for time-consuming tasks normally implemented in software. Whereas conventional processors are able to directly operate on data items of types such as byte, integer, and float, the basic data object on the R1000 is a 128-bit control word. The upper 64 bits of this control word contain an arithmetic value while the lower 64 bits contain a pointer to a type descriptor. During execution each of these 64-bit fields is routed to a dedicated arithmetic unit. The value unit performs the requested operation while the type unit simultaneously performs type checking. This allows two common activities, mormally done sequentially, to be done concurrently. To speed the transfer of these oversized control words, the main memory bus is 128 bits wide.

Another frequently occurring task is the packing and unpacking of data. This is commonly done in software by means of repeated shifts, ANDs, and ORs. Another component of the R1000 CPU, the field insertion unit (FIU) provides hardware control of this process by inserting or extracting the necessary data and pointer fields from the 128-bit control word. To further speed this process, a separate 64-bit bus connects the FIU to the value and type arithmetic units; this reduces the possibility of contention on the main memory bus.

A third major architectural enhancement involves the R1000's memory subsystem. Virtual memory management is traditionally handled by the operating system. Hardware support for this activity is generally limited to virtual address registers and a translation look-aside buffer which serves as an address cache, The R1000, however, performs the administrative tasks associated with page swaps and task management using microcoded hardware, thus providing yet another significant boost in performance.

System reliability was also an important concern in the design of the system and a number of steps were taken to address this issue. Proven, off-the-shelf peripheral devices were used and the workhorse DEC PDP-11 was incorporated as the I/O processor. The number of wired connections is reduced by using dense eight-layer circuit boards with foreplane and backplane connections. An additional microprocessor is installed on every CPU board to monitor board status, These monitor chips gather diagnostic data which is saved on system disks for later use. This information is used for quality control during the manufacturing phase and for preventive maintenance and fault location after the system is installed at a customer site. If an R1000 fails, an attached autodial modem calls the factory and transmits information which enables service personnel to bring the appropriate spares with them to the site. Additional general information on the R1000 may be obtained from (Rational 1992 Document D-80) and (Caruso 1985). R1000 system performance has been addressed in (Lamson 1991).

The Information Repository

The Ada Language Reference Manual requires that a certain amount of syntactic and semantic information generated by an Ada compiler persist beyond compilation time. This is necessary, for instance, to enable the compiler to enforce the integrity of interfaces when the associated packages and procedures are (re)compiled. Much of this information must be obtained in any case, either explicitly or implicitly, during the lexical analysis and parsing phases of compilation. Unfortunately, however, this data is typically discarded after the object code is generated and must be reproduced when a module is recompiled. The Environment avoids this inefficiency through use of the Descriptive Intermediate Attributed Notation for Ada (DIANA).

> DIANA is a intermediate form of Ada programs which has been designed to be especially suitable for communication between two essential tools—the Front and Back Ends of a compiler—but also to be suitable for use by other tools in an Ada support environment. DIANA encodes the results of lexical, syntactic, and *static* semantic analysis, but it does *not* include the results of *dynamic* semantic analysis, of optimization, or of code generation.[1]

As envisioned by its creators, DIANA is an abstract data type based on the mathematical model of attributed trees. It has a number of important properties. First, although the terms "abstract syntax tree" and "attributed parse tree" are often used in descriptions of DIANA, the notation is representation independent; i.e., instances of the the DIANA abstract data type do not necessarily have to be implemented using records containing pointers to other records. Second, DIANA is based on the formal definition of Ada; specifically, given the DIANA representation of an Ada program, it is possible to regenerate the original source

[1] From (Goos et al. 1983, p. 7); © 1978 Springer-Verlag, reprinted by permission.

code, except for comments. Finally, DIANA was designed to be both efficiently implementable as well as extendable; the existence of the Environment is probably the best evidence of the designers success in these respects. Further information on DIANA is available in the reference manual (Goos et al. 1983).

The information repository of the Environment may be viewed as an object database containing Ada programs represented using DIANA. A programmatic interface defines the specific operations which may be performed on database objects and thus protects them from uncontrolled access. Rational's own CMVC is built on top of this interface. The utility of the repository is increased by its ability to save objects of other types, including requirements specifications and design documents. The advantages to this centralized repository approach are several. First, availability of the programmatic interface encourages automation of many development activities. Second, the programmatic interface is methodology-independent, allowing use of whatever technique is appropriate. Third, the repository allows traceability from implementation back to requirements analysis and design; i.e., it is possible to determine which design decision motivated the construction of a particular Ada unit. Fourth, a centralized repository promotes consistency and thus aids in quality assurance. Fifth, the transition from one life cycle phase to the next is easier because the results of the current phase serve as the foundation for the next and are readily available in machine readable form. Finally, the standard programmatic interface to a single database of project information promotes compatibility with other software development tools (Rational 1988 Document TO-1), thus addressing the tool integration problem noted earlier.

Support for Basic Development Activities

This section will describe how a typical software engineer would use the Environment. Like UNIX, the R1000 operating system provides basic security functions; a user must supply a user name and a password to gain access to the system. Additionally, the system requests a ''session name'' to uniquely identify the current login; multiple sessions are allowed. The user is then presented with a screen divided into one or more work areas called ''frames;'' the default is three. Each frame consists of a ''major window'' and an arbitrary number of ''command windows,'' limited, of course, by the frame size. Major windows are used to display ''images'' of ''objects'' while command windows display commands for execution. The Environment recognizes several different types of objects, including files, Ada units, and libraries. Images of large objects may not fit within the window, so the window is actually a viewport presenting a portion of the image. Altering the image of an Ada unit, for instance, does not alter the underlying object it represents. Objects must be explicitly updated.

The Environment provides the customary functions to aid in window management; users may resize, reposition, or remove windows at their discretion. Windows themselves are treated as objects and are saved in a special window directory. When the Environment creates a new window (as a result of some user directive), an entry is created in this directory and the window image

is displayed in the least recently used screen frame. Only the image of a window is overwritten on the display, not the window object itself. Thus any window image may be recalled to the display at a later time. The Environment aids the user in managing this process by marking the next window to be replaced with a tilde (~) in the window banner and by allowing a window to be "locked" to prevent its image from being overwritten. Windows locked in this manner have their banners marked by the at sign (@).

As noted above, each major window has zero or more command windows associated with it. An Ada declare block is displayed in each command window and users may enter one or more Ada statements into this block. Incomplete command fragments may be automatically finished using [complete] [1] and then executed using [promote]. All activities, from resizing windows, to invoking the compiler, to interacting with the configuration management system may be accomplished via interactive execution of Ada statements. (This capability is essentially the Ada-based command interpreter noted earlier in the discussion of the full APSE.) Often, only a single procedure call is necessary to accomplish a given task.

Various keyboard aids are provided for the convenience of users; frequently used commands are bound to function keys to increase developer proficiency. Users may customize the keyboard by defining additional command-function key bindings. [help] followed by any function key will result in the display of a help window describing the task performed by that function key. Further information on function keys, commands, and tools may be obtained from the basic operations manual (Rational 1988 Product Number 4000-00116) and the user's guide (Rational 1988 Product Number 4000-00117).

The Environment provides an Ada-knowledgeable editor. As indicated earlier, the editor operates on an object image, not directly on an object. [promote] actually updates the underlying object and exits the editor; [enter] accomplishes the same function but allows editing to continue. All basic text editing operations including entering, deleting, moving, copying, searching, and replacing text are available. The real power, however, of the Rational editor is the result of two important features, formatting and semanticization. Formatting is accomplished using [format] and may be periodically performed during an editing session, even prior to the completion of a particular Ada unit. Formatting checks for syntax errors, finishes incomplete statements where it can, and where it cannot, supplies prompts. Function keys allow the user to walk through the prompts, allowing entry of additional code. As part of this process formatting cosmetically restructures the program text to conform to previously specified programming standards ("pretty-printing"). A simple illustration of the effect of formatting is given in Figure 5.

A second important capability is semanticization. As the name indicates, this function performs semantic checking; this includes verification of type

[1] This notation refers to a keystroke or selection of a menu option.

Before format

```
procedure calculate_statistics
is num_of_points : integer
```

After format

```
procedure Calculate_Statistics is
    Num_Of_Points : Integer;
begin
    [statement]
end Calculate_Statistics;
```

Figure 5. Example of the effect of format

compatibility, comparing the types of actual parameters with those of formal parameters, and detection of undeclared objects. Semanticization may even be performed on program fragments.

The Rational program development model is fundamentally different from the conventional model. The conventional model of program development is file oriented. A source file is created using an editor and translated by a compiler to produce an object file; this object file is then linked to produce an executable file. This process carries with it an inherent risk that an out-of-date version of an Ada unit will be inadvertently used or that a unit with only cosmetic modifications will be recompiled.

The Rational model avoids these problems by maintaining an Ada unit as a single object which exists in one of three states: source, installed, or coded. A unit in the source state is editable, but is not guaranteed to be syntactically or semantically correct. An installed unit is not editable in the usual sense; only certain restricted types of modifications may be performed on it. It is, however, syntactically and semantically correct and may be referenced, or "withed," by other Ada units. A unit in the coded state also has this consistency and referenceability, but unlike an installed unit only its specifications may be modified. Importantly, this last state is the only one for which machine code is generated.

From a user perspective, the mechanics of performing these transitions is quite simple. Assuming the Ada unit is syntactically and semantically correct, successive applications of promote will move a unit from the source state to the installed state to the coded state, and finally execute it. demote will migrate a unit in the reverse direction. It is also possible to explicitly move one or more units to a particular state. More specifically, there is the capability to promote all units in a library to the coded state with a single command, in which case the

Environment handles all compilation dependencies and maintains the program's overall semantic consistency.

From a system perspective, transition from the source to the installed state requires that the DIANA representation of the Ada unit object be created, while transition from the installed to the coded state requires that a machine code representation be generated. In conventional terms, these two transitions are equivalent to the processing performed by the front and back ends, respectively, of a compiler. Importantly, however, the associated DIANA tree and machine code are viewed not as separate files, but as information associated with a single object.

An important feature of the Environment is the ability to "browse" through a program. While editing or debugging a particular Ada unit a software engineer often needs to know a variable's type definition, to determine the purpose of a subprogram, or to find other locations where a variable or unit is referenced. When using a conventional system this requires searching through one or more files using utilities such as `grep` and `find`. Using its DIANA-based program representation, the Environment provides a much more satisfactory and convenient mechanism for accomplishing such tasks. Moving to the enclosing Ada unit is performed using [enclosing]. [other part] allows a developer to move back and forth between a unit's specification and body. The definition of a particular program structure may be located by first selecting the item of interest, such as a variable reference, and then using [define] to display the item's definition. Program structures may be easily selected using [object] in conjunction with the four arrow keys; this process is best understood in the context of the DIANA tree representation of an Ada unit. Using [object] [←] moves toward the root, selecting larger and larger program structures, while [object] [→] moves away from the root, focusing on more and more specific program fragments. [object] [↑] and [object] [↓] allow lateral movement to neighboring branches. For example, if the second statement of a block is currently selected, then [object] [↑] (up the screen) will select the first statement while [object] [↓] (down the screen) will select the third. These browsing and selection features are particularly valuable when the software system is large and complex.

The Environment debugger is powerful, yet easy to use. A developer merely turns on debugging mode; recompilation is unnecessary. The debugger allows execution to proceed until a fault or checkpoint is encountered. Single-step execution, establishment and removal of checkpoints, display of variable values, and display of the call stack, are all possible. The browsing facility may be used in conjunction with the debugger to move back up through the calling sequence and locate the source of a problem. Additionally, the debugger keeps a log of user interactions which may be saved for later inspection.

Because conventional systems do not take the syntax and semantics of a program into consideration, a seemingly minor modification to a simple program unit may require recompilation of many other units, and of course the modified unit itself must be recompiled in its entireity. Many such systems use only time stamps to determine system consistency, and so even cosmetic changes

involving indenting or comments require unit recompilation.

The Environment avoids much of this work because programs are represented using DIANA trees. Because the Environment manages dependencies at the statement level, the impact of a change is greatly reduced. Furthermore, DIANA enables incremental compilation of individual statements or declarations, thus avoiding recompilation of an entire unit. Specifically, it is possible to incrementally add, change, or delete statements and declarations without dependencies in unit bodies in the installed state and in package specifications in the installed or coded state. Comments in any installed or coded unit may be manipulated in any fashion. Modifications to declarations with dependencies may still require significant recompilation, but even then the system will provide notification of the potential impact of the impending change, thus allowing the developer to make an informed decision about proceeding any further.

Subsystems

As noted earlier, the Environment is written in Ada and is the APSE used for its own development and maintenance. As the Environment grew in size, the nature of the problems developers encountered differed significantly from those arising in smaller projects. Specifically, it became increasingly difficult to limit unplanned dependencies between units. This situation, termed "design degradation," increased compilation time and the additional program complexity made the code difficult to maintain. Furthermore, this complexity was preventing multiple development teams from working in parallel, resulting in a loss of productivity. Finally, as the project grew to several hundred thousand SLOC, these problems slowed development to a crawl and the project entered a "thrashing" mode. This phenomenon is not unique to the Environment; anecdotal evidence indicates that other large projects have encountered similar problems at approximately the same number of SLOC. It became clear that a level of abstraction higher than the Ada package was necessary and that tools would have to be provided to support this abstraction. Subsystems provided a solution to these problems.

Some discussion of Ada system structure is necessary to understand the subsystem concept. Figure 6 shows the topology of a hypothetical Ada system. Ada compilation units are represented by nodes having a specification, or "spec," (unshaded) and a body (shaded), while arrows represent dependencies (parents depend on children). Note that the origin of an arrow is significant; either the specification or the body of a unit may depend on the specification of another unit. Typically, when a unit is modified, it must be recompiled, along with any other units which directly or indirectly depend on it. Figure 6 also illustrates this idea by lightly shading the units which must be recompiled as a result of the indicated modification. Such cascading dependencies often result in lengthy recompilations. Subsystems help limit the impact of such changes by providing a higher level mechanism for grouping Ada units together.

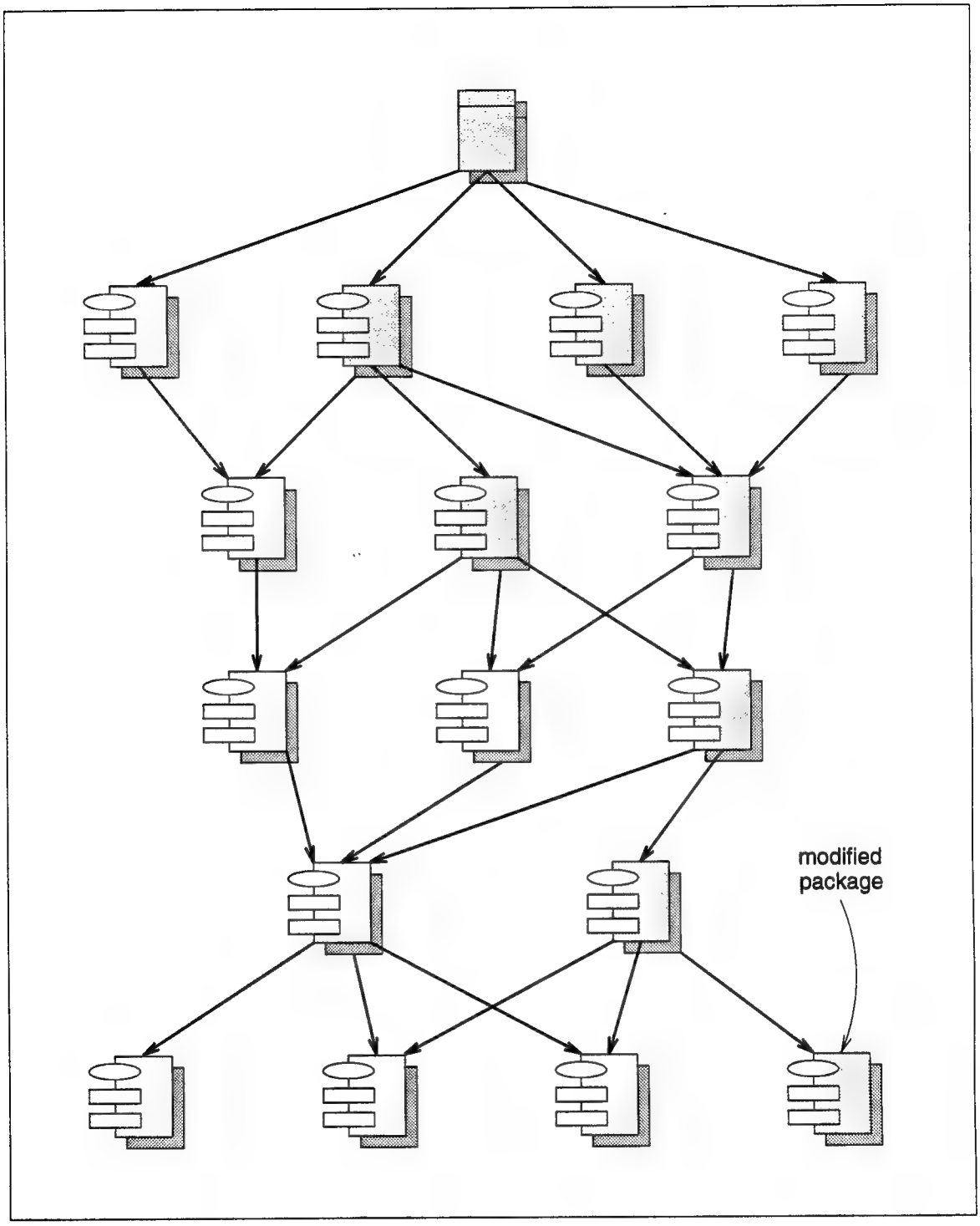


Figure 6. Recompilation required by cascading dependencies

Like all other entities, subsystems are treated as objects by the Environment. A subsystem object may be thought of as a "superpackage" with many of the same attributes and features as a package. Subsystems have specifications (or exports) which list the resources (e.g., packages) available for use by other subsystems. Subsystem imports list the other subsystems on which this subsystem depends. The implementation portion consists of the actual Ada code implementing the subsystem's features and resources; it is analogous to a package body, but may also contain design documents, test programs, and other related material. Finally, a subsystem contains history information recording when the subsystem was created, modified, compiled, and released.

Subsystems have proved to be a powerful mechanism fo addressing problems that arise in large Ada system development. Because the Environment provides for separate compilation of subsystems, they may serve as "firewalls" which limit the scope of a recompilation. If the changes are restricted to a subsystem body, then only units in that subsystem will be recompiled. This has the potential for reducing compilation time and increasing developer productivity. Subsystems reduce the tight coupling between the various components of a large software system, and thus promote the work of parallel development teams. Finally, subsystems provide a coarse-grained building block for large system design, a benefit which is enhanced through the use of graphical design tools (to be discussed later). Further information on the use of subsystems in the context of large system development may be found in (Rational 1988 Document 6004).

Configuration Management and Version Control

Management of a software development effort is in many respects similar to the management of other product development activities. A number of managerial and technical issues must be addressed to ensure success, including breaking the project into units of manageable size, specifying how these units join together, actually completing the units, testing the units individually and as a whole, releasing new versions of the product, and coordinating the efforts of the development team (Rational 1988 Product Number 4000-00129, p. PM-1). Large projects of any variety are inherently complex and require commensurate amounts of time, material, and personnel to complete.

This complexity and the burdens it imposes on a development team have already been noted for the case of large software projects. However, two aspects of software development efforts make them unique. First, like a hammer on a fragile statue, a very localized change may have drastic and unfortunate side effects. Stated another way, large software systems seem to be more "brittle" than other products of similar size. Perhaps this is why the production of such systems is often treated as an art, and why, even after several decades of of effort, attempts to impose a discipline on the process have been less than successful. To use the statue analogy, software is too often sculpted instead of engineered.

A second differentiating aspect of software development is more positive. Because computers are unavoidably involved in the process, it is convenient to use them to automate the management of a project's complexity. In fact, it is possible to use automation to one extent or another to address all of the managerial and technical issues noted at the beginning of this section. Brooks quite accurately refers to this capability as "an indispensable technology."[1] In the case of the Environment, the features that actually enable this automation of project management tasks are subsystems, which have already been discussed, and a set of capabilities collectively termed CMVC.

Perhaps the most basic function provided by CMVC systems, including Rational's, is controlled access to objects. The library terminology typically used to describe this access provides an accurate analogy. An object (e.g., an Ada unit), like a library book, is "checked out," guaranteeing the user sole access to that object. When a developer has completed his modifications, the object is then "checked in." Without this capability, multiple users could simultaneously access and modify the same object, possibly overwriting one another's work.

Rational's CMVC, however, goes far beyond this minimal capability. Any object managed by CMVC is said to be "controlled" and the Environment associates a "reservation token" with each such object. Checking out the object is accomplished by acquiring this token. Information on controlled objects, including the status of the reservation token, is maintained in a "CMVC database;" there is one such database for each subsystem. When a user checks out an object, modifies it, and checks it back in, the Environment creates a new "generation" of the object in the CMVC database (Rational 1988 Product Number 4000-00129, p. PM-6). Generations are stored as "negative deltas;" i.e., the Environment maintains a complete copy of the most recent version of an object, while older versions must be reconstructed using the saved differences (deltas) between successive generations.

Associated with each subsystem are one or more Ada "program libraries" containing the Ada units belonging to that subsystem. A "configuration" is a collection containing one generation of each controlled object in a program library. Typically, that particular collection containing the most recent generations of all objects is called the "working library." (Brooks refers to this as the "playpen." (Brooks 1975, p. 133)) It is from this configuration that developers check out objects for modification, compilation, and testing. There must be at least one such working library per subsystem.

When all of the units in a working library compile and developers are satisfied with the library's status, a "frozen" copy of the working library may be created. This copy, also called a "release," is a complete compiled program library. It may be thought of as a "view" or "snapshot" of the current state of the project. Rational refers to a succession of such releases generated from a

---

[1] From (Brooks 1975, p. 133); © 1975 Addison-Wesley, reprinted by permission.

particular working view as a "development path." This concept is illustrated in Figure 7.

This situation becomes more complicated when a project involves multiple subsystems. The released and working views noted above are "load views" containing complete implementations of all subsystem components. However, it is not always necessary to have such a full implementation; as an example consider the following situation. When subsystem S1 references (or "imports") resources from subsystem S2, then S1 requires only interface information about S2 in order to compile. This information is provided by a skeletal "spec view" of S2 which specifies the resources S2 is willing to share (or "export") to other subsystems. Use of this feature on an actual project will be seen in a subsequent section.

Another benefit of Rational's CMVC is related to the integration of programs having multiple subsystems. Although it is generally true that the current release of a program is a combination of the current releases of each of its subsystems, it is also often necessary to assemble custom releases of the program as well. This may be necessary when developing customer-specific versions, or when developers want to use older, perhaps more reliable, releases of some subsystems to track down problems during testing. This is accomplished by means of "activities." An activity may be viewed as a table specifying which release of each subsystem to use when creating an executable program. Building a custom version thus involves merely defining a new activity and then directing the Environment to assemble the precompiled views specified therein.

Rational's CMVC has other distinctive features which enhance developer productivity. Descriptive text, or "notes," may be associated with every controlled object to provide additional information. Better project management is promoted by the use of "work orders" to assign development tasks. When they are used, the Environment automatically logs any CMVC commands executed in response to a particular work order (Rational 1988 Product Number 4000-00129, p. PM-15). Work orders may also be customized to suit the needs of a particular project.

For an Environment-specific overview of configuration management, see (Morgan 1988); more general information may be found in (Dart 1990, Dart 1992).

Future of the Rational Environment

What is the future of the Environment? One of its most serious shortcomings has been the proprietary hardware required to support it. Admittedly, this hardware may also be viewed as one of the Environment's strengths, and the reasons behind the development of that hardware have already been presented. Nevertheless, increases in microprocessor performance, particularly those with reduced instruction set computer (RISC) architectures, coupled with Rational's view of itself as first and foremost a provider of software engineering products

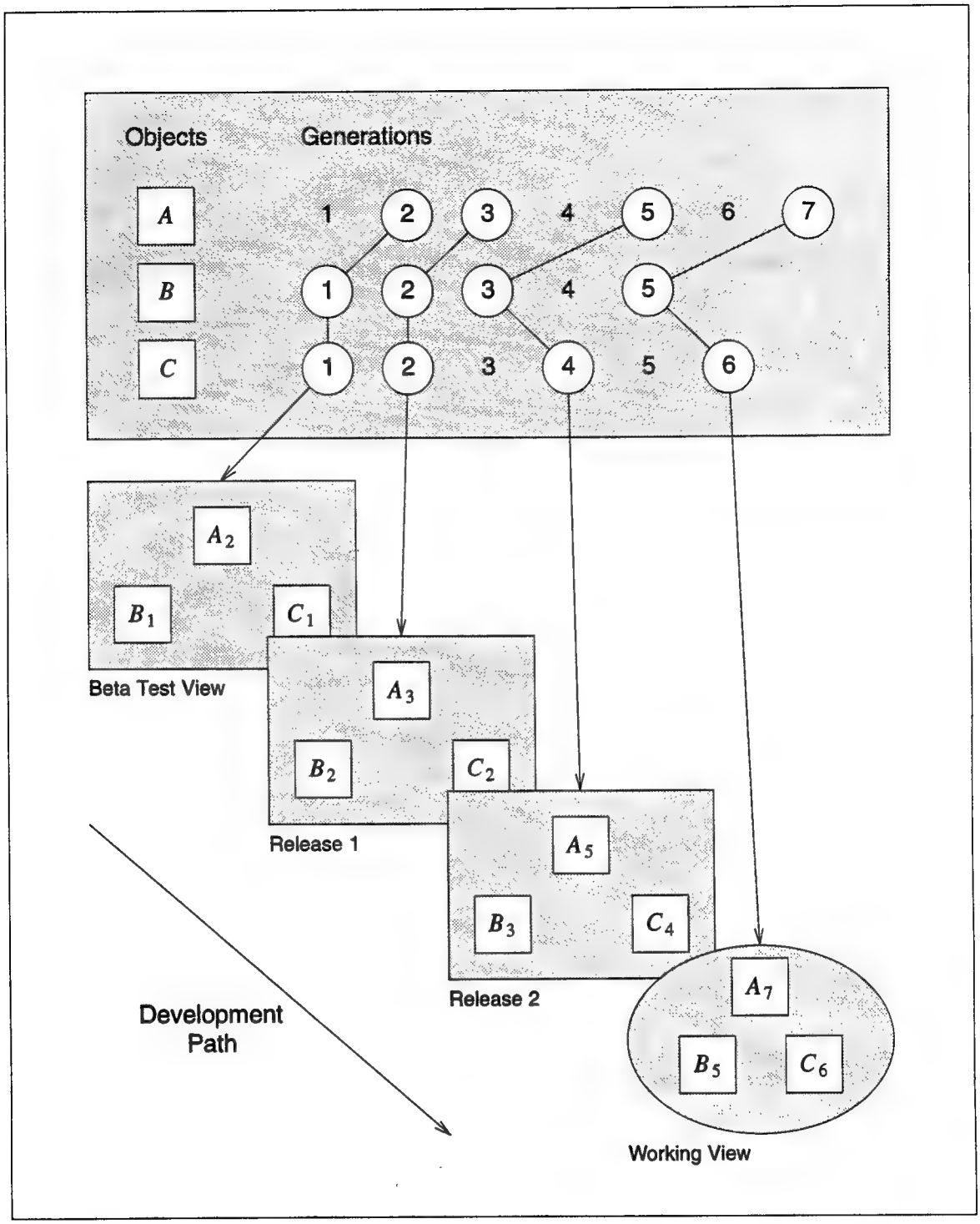


Figure 7. A development path (adapted from (Rational 1988 Product Number 4000-00129, p. PM-9))

and services, led the company in 1991 to begin development of an "open-systems" version of the Environment. Apex, as this new version is called, was introduced in August 1993 and is currently available on two RISC platforms: IBM RS/6000 and Sun SPARC.

Apex contains all of the key software engineering features of the proprietary-platform version of the Environment. The persistent intermediate representation based on DIANA is still the key to its unique functionality. The capabilities for syntactic and semantic completion within the editor, hypertext browsing of programs, integrated debugging, and CMVC are all present in the new product. Moreover, Apex is more than just a port of the Environment to a UNIX platform. Several enhancements, including replacement of the R1000 incremental compilation technique with "optimal recompilation," an improved software release mechanism, and support for mixed language programs (e.g., Ada and C++), make Apex a true next generation software engineering product.

Apex is not a closed, proprietary environment built on top of UNIX. Whenever possible, Apex developers used capabilites provided by UNIX, the X Window System (X), and the Motif window manager. Unlike the R1000 Environment, APEX does not include a file system, window manager, mail handler, security and accounting facilities, or system administration functions, because those features are already supplied by the operating system. It is a true Motif application which uses standard Motif menus, dialog boxes, and customization options. For example, the Motif text widget was used to construct the Apex Ada editor. Furthermore, users may control the appearance of the Apex interface just as they do other Motif-conformant applications by assigning their own values to the appropriate X resources in their `.Xdefaults` file. Additionally, all of the Apex functionality delivered through Motif is also available through a UNIX command line interface, thus allowing programmatic access through the shell. Apex CMVC is layered on top of the UNIX Revision Control System (RCS). Many of the problems related to managing large projects on multiple R1000s (Blair 1992) are resolved under Apex through use of the Network File System (NFS). This allows large programs formerly split across several machines to be logically present in their entirety on each of a team's file servers and workstations.

Like its R1000-based predecessor, Apex is a resource intensive application. Surprisingly, however, processor speed is not a performance bottleneck. "Rational Apex, even on a moderately powerful RS/6000 or SPARC workstation, outperforms the Environment on the R1000." (Amos 1993, p. 7) However, users must plan to provide sufficient memory; Rational recommends a minimum of 32 Mbytes of memory for a single-user workstation. Instead of purchasing additional workstations, managers may decide to cut costs by accessing the Apex workstation via cheaper X terminals or PC-based X terminal emulators. If one of these options is selected, Rational recommends an additional 32 Mbytes of memory for each additional user. Disk storage is also an issue. The Apex program itself requires about 150 Mbytes of disk space; to this figure must be added whatever space is needed for Ada application development. Rational's rule of thumb for estimating this is 1 Kbyte per SLOC; thus a one million SLOC

application would require about 1 Gbyte of secondary storage.

In spite of the many attractive features of Apex, Rational realizes that migration of many of its R1000-based customers will be dictated by project-related and budgetary constraints. Additionally, many of Rational's layered products will not be available on Apex until 1994. For those reasons, Rational is continuing to support and enhance the R1000 product and is providing mechanisms for the two systems to run in parallel so that the user transition will be smooth.

3 Tools to Augment the Rational Environment

This section describes several tools which enhance the functionality of the Environment. Some of these tools provide front-end CASE capabilities so that the resulting environment covers additional life cycle phases. Not all of these tools may be applicable to a particular project; guidance for determining if particular tools should be acquired may be obtained from (Zarrella, Smith, and Morris 1991).

Rational Design Facility

The Rational Design Facility (RDF) is a CASE tool which runs under the Environment. It allows project managers to specifiy standards for system design and implementation and then provides for the automatic enforcement of those standards. Two concepts must be presented to understand the capabilities of the design facility: methodologies and program design languages (PDLs); these will be discussed before further describing this tool.

A primary goal of software engineering is to discover better ways to develop software. To reach that goal, numerous step-by-step development approaches, or "methodologies," have been proposed. There are important differences between many of these, e.g., consider object-oriented versus structured methodologies. Nevertheless, most methodologies have the following features in common (Rational 1989 Product Number 4000-00362, p. I-1).

- The software development process is described using a *life cycle model*, e.g., waterfall or spiral.
- This model is partitioned into *life cycle phases*, e.g., system analysis, requirements analysis, preliminary design.
- *Design elements* are used to reason about the design.

- *Design standards* specify the valid ways in which design elements may be used.

- *Implementation standards* provide rules governing software creation.

- Various stages of the development process require production of *documents*, e.g., software requirements specification, interface design document.

- Finally, the methodology specifies the *relationships* between these components.

The second important concept is that of a PDL. A program design is made up of design elements, and these elements are specified using a PDL. PDLs are more structured than natural languages, but tend to be less structured than actual programming languages. Generally, the more structure a PDL has, the easier it is to automate various design activities. Some PDLs are graphical in nature; this increases their intuitive appeal, but unless accompanied by some sort of supporting text, it is difficult for them to capture the system design in a detailed way.

The PDL used by the RDF is made up of structured comments termed ''annotations,'' and various Ada statements called PDL ''elements;'' a fragment of PDL is shown in Figure 8. Although the elements could be any type of Ada statement, allowable elements are typically restricted to some subset of Ada. This subset typically includes, for instance, package and procedure statements. The RDF uses design rules to enforce adherence to this allowable subset; design rules may be customized as needed.

The annotations used in the RDF PDL are denoted syntactically by the concatenation of the Ada comment symbol (--) and the vertical bar (|); this combination is the ''annotation symbol'' (--|). Annotations may be one of two types. A ''simple annotation'' consists of one or more lines of text, each beginning with the annotation symbol. A ''key word annotation'' also begins with the annotation symbol, but is followed by a key word; this key word must begin with the at symbol (@). Keywords may be followed by zero or more arguments. Annotations of both types may be ''attached'' to PDL elements merely by placing them adjacent to one another. Blank lines serve to ''detach'' an annotation from an element. An individual annotation is terminated by a PDL element, a following key word annotation, a blank line, or a blank annotation line.

A DOD-STD-2167A-compliant design must begin with a *system* component, which is then broken down into zero or more *segments*. The design process continues by expressing these highest level components as *hardware configuration items* (HWCIs) and *computer software configuration items* (CSCIs). Each CSCI must consist of one or more *computer software components* (CSCs). Each CSC is further refined into either additional CSCs or into the atoms of the process, the *computer software units* (CSUs). When using the RDF to design such a system a design team first creates a PDL description of each of the high level design components, beginning with the system (root) component and continuing through the HWCI/CSCI levels. This high level design is then iteratively refined by

```
--| @COMPONENT_KIND          CSCI
--| @ID                      5
--| @ABBREVIATION            CSCI_LSS
--| @DOCUMENT_NUMBERS       (SRS    => R-02-LSS-47A,
--|                          PSDD   => D-09-LSS-22C,
--|                          SDD    => D-09-LSS-23B)
--| @CAPABILITIES           (Initialize_Life_Support_Environment,
--|                          Monitor_Life_Support_Environment,
--|                          Maintain_Life_Support_Environment)
--| @EXTERNAL_INTERFACES    (Life_Support_System_Status_Display,
--|                          Life_Support_System_Controls,
--|                          Life_Support_System_Alarm)
--| @INTERNAL_INTERFACES    (Central_Hvac_System_Control,
--|                          Compartment_Air_Duct_ControlS,
--|                          Compartment_Biohazard_Detectors,
--|                          Compartment_Pressure_Gauges,
--|                          Compartment_Radiation_Gauges,
--|                          Compartment_Temperature_Gauges)
--| @INTERFACES_USED        (Backup_Power_Supply_Status_Gauge,
--|                          Bulkhead_Door_ControlS,
--|                          Compartment_Biohazard_Sterilizers,
--|                          External_Radiation_Scanners)
--| @PURPOSE This CSCI is the life support system for the manned
--|          Mars probe. As such, it both monitors and maintains
--|          a suitable environment for the probe's crew.
--| @STATES     (Startup    (Power_On, Start_Display, Pressurize),
--|              Automatic  (Monitor_Status, Maintain_Status),
--|              Manual     (Accept_Input, Alter_Environment),
--|              Shutdown   (Depressurize, Stop_Display, Power_Off))
--| @ALLOCATION (Capability (Initialize_Life_Support_Environment,
--|                          Monitor_Life_Support_Environment,
--|                          Maintain_Life_Support_Environment))
--| @DECOMPOSITION          (LSS_Display_Subsystem,
--|                          LSS_Controls_Subsystem,
--|                          LSS_HVAC_Control_Subsystem,
--|                          LSS_Air_Quality_Monitoring_Subsystem,
--|                          LSS_Radiation_Monitoring_Subsystem)
package Life_Support_System is
end Life_Support_System;
```

Figure 8. RDF PDL for a hypothetical CSCI

producing PDL components for CSCs and CSUs until a detailed design is produced.

Throughout this design process the RDF assists the designer in a number of ways. It provides PDL templates for the various design components and prompts the designer for the appropriate keyword annotation arguments. These arguments are then checked for type consistency. edit opens the library package specification containing the PDL for a particular hierarchical component, e.g., a CSCI. PDL complete is then used to supply the annotations required by the current life cycle phase. Annotation arguments are then provided by the user; PDL format may be used to perform syntactic completion of these arguments. Finally the Ada portion of the PDL must be entered; PDL format may be used here as well. The RDF checks for design consistency by verifying that the design is a valid hierarchy of components of the appropriate types. Interface consistency is also checked by examining the `@INTERFACES_USED` and the `@EXTERNAL_INTERFACES` annotations in CSCI components.

One of the most powerful features of the RDF is its document generator. This allows construction of hypertext documents which are linked to PDL annotations, text files, and graphics files. The ability to produce required design documents by extracting the appropriate annotations provides a single point of maintenance when a document must be revised. Furthermore, any changes in the design are immediately reflected in the documentation. Naturally the browsing capability of the Environment is quite helpful in this context, allowing traversal between PDL and related documents.

The RDF is a powerful design aid because of its universality and flexibility. Designers may customize the design rules and rely on the RDF to automatically enforce those rules. Finally, because the PDL is actually a subset of Ada, it is always compilable and allows the design to naturally evolve into the implementation. Further information on its use may be found in (Rational 1992 Document D-79) and (Ripken 1988).

Rational Insight

Rational Insight is a front-end CASE tool which addresses the design phase of the software life cycle. It may be used to browse software systems residing on the Environment, to reverse-engineer such systems, to design a new software system, to experiment with alternative designs for existing systems, and to print design diagrams for inclusion in system documentation. Insight has a Motif-compliant GUI which enhances its usefulness.

The two primary components of Insight are the Data Manager and the Illustrator. The Data Manager is a server which extracts information about software systems; it runs under the Environment on an R1000. The Illustrator, on the other hand, is responsible for generating diagrams; it runs as an X application on a UNIX workstation. Additionally, to invoke Insight, a user must have a device to access the Environment and an X display to open Insight windows. The

access device and the display may be part of the same computer system. For instance, a user may install the Rational X Interface (RXI) on the same UNIX workstation that runs the Illustrator; in this case the workstation monitor serves as a local X display. It is also feasible to use a second UNIX workstation, an X terminal, or a PC-based X terminal emulator as a remote X display device. Finally, it is also possible to run Insight in batch mode without an X display; this is convenient when one only wishes to print a diagram.

Insight is started using the `Insight.Invoke` command. Arguments to this command specify where in the file hierarchy the software system to be analyzed is located, on which machine the Illustratior is running, the name of the X display, as well as other information. Entering this command followed by promote causes the Insight main window to appear on the X display. At this point a user may use the File menu to, for example, select an existing diagram for viewing, or to initiate creation of a new diagram file. Other menus provide browsing capabilities and permit selection of various tool options.

As a brief example of the use of Insight, consider the following scenario. A developer has started Insight, indicating the appropriate position in the file system as the current object context. Wishing to browse a particular group of subsystems he sets the current "activity" to `interface_activity`, which is treated as a relative pathname to the current context. After selecting File:New and interacting with the File:New dialog box, a diagram window appears on the display. The title bar of this window contains the diagram file name. The window itself may be manipulated like that for any other Motif application; it may be opened, closed, resized, and moved in the usual ways. Inside the window border is the diagram canvas. Horizontal and vertical scrollbars allow positioning the viewport on any portion of the canvas, a necessary feature for large diagrams. The diagram itself is made up of icons denoting program objects and arcs indicating dependencies among those objects. The icon notation is adapted from that specified in (Booch 1987, pp. 55-59). There are different icons (node symbols) to represent a subsystem, library, group, main program, subprogram specification, subprogram body, generic subprogram, package specification, package body, and generic package. In this example the subsystems contained in `interface_activity` would be presented as subsystem icons joined by dependency arcs. To browse a particular subsystem, the user clicks on the corresponding subsystem icon and selects Browse:Diagram. At this point a second diagram appears which illustrates the various packages in the selected subsystem along with their various dependencies. Using the mouse to select a particular package body icon, followed by Browse:Environment will then display the text of that Ada package in a separate window.

Insight has other useful capabilities which allow it to be used as a design aid at the beginning of a large development effort as well as a reengineering tool throughout the course of a project. A complete discussion of these capabilities is beyond the scope of this report; see (Rational 1992 Product Number 4000-00676) for further information.

Cadre Teamwork

Cadre Teamwork is an integrated software development system which supports the analysis, design, and coding phases of the software life cycle. Teamwork may be used in a stand-alone mode or in conjunction with a variety of other tools and compilers, including the Rational Environment. Of particular interest here, however, is that Rational has provided more than just a mere communications protocol between the two environments. Rather, Rational has constructed an interface integrated into the Environment which allows developers to make a smooth transition from use of Teamwork's graphical design aids to Ada source code implementation.

One of Teamwork's aids is Teamwork/SA, an intelligent editing system which runs on UNIX workstations. This system is actually made up of three tools: a data flow diagram (DFD) editor, a process specification (P-Spec) editor, and a data dictionary entry (DDE) editor.

The DFD editor has the usual mechanisms for interactive creation and modification of data flows, data stores, process bubbles, terminators, text, and labels. The P-Spec editor is used to produce text describing how input is transformed into output by a process. It automatically generates the the title and I/O list from the parent DFD. As DFDs are created, Teamwork/SA automatically creates a template DDE for each data flow and data store. Each of these DDEs consists of attribute types and definitions which are completed by the user with the DDE editor. Teamwork/SA also includes a facility for automatically checking various aspects of the system specifications, such as errors in syntax and consistency between DFDs, P-Specs, and DDEs.

Cadre also provides extensions to Teamwork/SA to facilitate production of specifications for real-time systems: these include control specification (C-Spec) editors for state transition diagrams, state event matrices, process activation tables, and decision tables. Together these enhancements are called Teamwork/RT.

After system specifications have been developed using these tools, the Rational-Teamwork interface enables extraction of the information necessary to produce the various requirements documents mandated by DOD-STD-2167A, e.g., software requirements specification (SRS) and interface requirements specification (IRS). Both the SRS and IRS are written in RDF PDL, so that the Rational-Teamwork interface actually generates PDL from the annotated diagrams produced using Teamwork/SA. Furthermore, access to these diagrams may be managed by placing them under the control of CMVC.

A second aspect of the Rational-Teamwork interface addresses the design phase of a project. Teamwork/Ada, also workstation-based, provides an editor for creating Ada structure graphs (ASGs). These ASGs are specified using the Buhr notation (Buhr 1984) which provides graphical design elements for a variety of Ada entities, including packages subprograms, generics, exceptions, and tasks. The ASG editor allows opening multiple windows containing

different views of the same diagram with changes made in one window immediately propagating to the others. This enables ASG modification from simultaneous high-level (zoom out) and detailed (zoom in) perspectives.

After the design has been specified using the ASG editor, the Ada Source Builder is applied to create compilable Ada code skeletons from the ASGs. The Rational-Teamwork interface is then used to import these skeletons into the Environment for further manipulation by the RDF.

An alternative to immediately importing the code to the Environment is to instead use Cadre's Design Sensitive Editor (DSE). The DSE is similar in spirit to the Rational editor in that it is syntax-directed. Unlike the the Rational editor it does not perform semantic analysis; however, it is knowledgeable of the ASG design on which the code is based and is therefore able to maintain consistency between the two. The DSE may be customized to give it a Rational "look-and-feel," thus providing a consistent interface to the user. Cadre recommends that the DSE be used for further code modification even after it has initially been imported to the Environment in order to preserve design-implementation consistency. Further information on the integration of the Environment and Teamwork may be found in (Cadre 1991 Application Note C-ANR) and (Rational 1988 Product Number 4000-00602). Additional descriptions of Teamwork capabilities may be found in Cadre product brochures (Cadre 1991 Publication C-PBTADA, Cadre 1991 Publication C-PBDSE, Cadre 1991 Publication C-PBSA).

Rational TestMate

Testmate assists software developers, test developers, and integrators in the testing and validation of a software system. It may be used to automate tasks associated with unit testing, integration testing, regression testing, and target testing. Testmate may also be customized by software toolsmiths to provide additional functionality.

Unit testing (the testing of individual program modules) generally involves construction of a driver program to call the module under test (MUT). Stubs for any subprograms called by the MUT must also be supplied. The development of this additional overhead software, or "scaffolding," (Brooks 1975, p. 148) may be sufficiently involved as to require delay of complete module testing until the integration phase. Other types of testing may also require such overhead.

Testmate facilitates the construction and use of this scaffolding using an object-oriented approach. The objects manipulated by Testmate during the testing process include "test cases," "test sets," "test scripts," and "test results." Instances of test case objects are implemented using a test case file which contains all the information associated with a specific test, including a description, driver, input files, output files, test context, set up and clean up routines, guidelines for interpretation and logging of results, and time constraints. These test case files are created using the Testmate test case editor; there must be one such file for each test. Test sets are collections of related test cases; a test case may be

a member of multiple test sets. Using information in the test set, Testmate automatically generates an executable Ada program called a test script which performs all of the tests in the test set. The output obtained from running this script is termed a test run file, and includes pass/fail status for each test case, a log of the test run, optional information on test coverage, and other information. Testmate provides special purpose editors which allow developers to browse this output.

Test results may be analyzed in a number of ways. The user-written driver may examine and verify the MUT's output parameters. Alternatively, test case output may be directed to a file which is then compared to the correct output. If such a file comparison is not possible, the user may provide a routine to examine the output file and determine if it is acceptable. Finally, if none of these options is feasible, manual inspection is allowed. In the first and third methods listed above, the user-supplied code must call the Testmate Ada routine `Tms.Register_Result` to report pass/fail status back to Testmate for recording in the test run file. In the second case, Testmate records this result automatically. In the last case, `Tms.Register_Result` must be interactively invoked.

Another possible by-product of a test run is test coverage information indicating how much of the code was actually exercised in the test. The default is to gather coverage information for every test case, but Testmate allows users to tailor this data collection capability and restrict data gathering to selected test cases, units, directories, or worlds. It is also possible to display the Ada source code of the MUT with untested coded segments so marked.

Testmate also has the ability to display and compare information from multiple test runs. Regression testing is facilitated by comparing output from two runs of the same script. When the output has changed, the version information recorded by Testmate allows a developer to track down the source code modification causing the output change. Using the Rational Compilation Integrator (RCI), it is also possible to migrate tests from the R1000 host to the target platform and then use Testmate to repeat the testing there in an automated fashion. Finally, as with other Rational tools, Testmate package specifications are available to development teams so that they may customize Testmate to meet their particular needs. Additional information on the use of Testmate may be found in (Rational 1992 Document D-82) and (Rational 1992 Product Number 4000-00720).

Other Tools

Rational provides other products to address various aspects of the development process. To promote the production of high quality documents, Rational has provided an interface to the Interleaf TPS desktop publishing system, as descibed in (Rational 1989 Product Number 4000-00334). At some point in a development effort, testing must migrate from the R1000 to the target platform. Rational's Target Build Utility (TBU) facilitates this migration process by

automating transfer of Ada units to the target and generating the necessary job control script to compile and link those units. The TBU is described in (Rational 1992 Product Number 4000-00375). The RCI addresses this same problem in a slightly different manner. It allows control of Ada compilation on a remote target machine from within the local host R1000 Environment and helps maintain program library consistency between the two platforms. Further information on its capabilities may be obtained from the user's manual (Rational 1992 Product Number 4000-00500).

There are other third-party tools which are also well integrated with the Environment. Adamat is an Ada code measurement and analysis tool which collects and analyzes various software metrics; see (Dynamics Research Corporation Systems Division 1992) and (Levine, Anderson, and Perkins 1990) for further information.

To facilitate the development of large software systems in the command, control, and communications domain, TRW has developed its Universal Network Architecture Services (UNAS). UNAS includes reusable Ada software components, as well as tools to assist in the development and instrumentation of Ada code (Royce et al. 1991). TRW has further extended the capabilities of UNAS through the Software Architect's Lifecycle Environment (SALE), a knowledge-based aid which automates various aspects of the design phase of a project (Royce and Brown 1991). There are R1000 versions of both UNAS and SALE.

4 User Experience

This section illustrates the effective use of the Environment by presenting case studies of its use on four projects: CelsiusTech and FS 2000, Computer Sciences Corporation and STANFINS-R, IBM and WISCUC, and Statistica and SIDPERS-3. Also included are reports of two evaluations by the Software Engineering Institute (SEI) and Magnavox Electronic Systems Company. Additional users are noted at the end of the section.

CelsiusTech and FS 2000[1]

CelsiusTech[2] is a major Swedish supplier of defense-related real-time systems. Over the fifteen years prior to the initiation of the FS 2000 project discussed here, CelsiusTech produced about 25 such systems ranging in size from 30,000 to 700,000 SLOC. The latter required seven years and about 300 man-years to complete.

Realizing that their existing software engineering methodologies were reaching the limits of their effectiveness and that future projects would be larger and more complicated, CelsiusTech decided to consider use of Ada on the Environment as an alternative to their current techniques. Their pilot test of this new technology, as well as their first Ada project, was UndC, a mobile command and control application for the army; it would be hosted by a DEC MicroVAX installed in a van. This project began in August 1985 and had an estimated delivery cost of $2.16 million. In spite of their lack of Ada experience and the inevitable requests for additional system features, this application was completed in December 1986 at a cost of $760,000, roughly one-third of their original estimate.

---

[1] The primary sources for this section were (Rational 1991 Document CS-1) and (Bachman and Marasco 1992).

[2] Since 1987 this company has changed its name from Philips Electronikindustrier to Bofors to NobelTech to CelsiusTech.

This positive experience with the Environment led CelsiusTech to use it on the then recently awarded FS 2000 contract. FS 2000 is an embedded shipboard command, control, and communications application integrated with various types of weapons systems. The software would be delivered on ships from five different countries, and depending on the configuration, between 150 and 600 programs are installed on each ship. The complete software package, targeted for the Motorola MC68020, consists of about 1.5 million Ada SLOC.

Delivery of FS 2000 on different ships in a variety of configurations required flexibility of design and customization of features. Key aspects of the software development process included emphasis on reusability, management of geographically separated teams of software engineers, and use of object-oriented design. The engineers assigned to this project were fairly well acquainted with use of structured programming in a high-level language: however, prior to the UndC and FS 2000 projects, they had no experience with Ada, software tools, and software development environments. Rational worked with CelsiusTech to provide a customized six-week training program which was carried out over a three to five month period. Small class size (twelve students) and a substantial hands-on component (40 percent of the course) proved to be very effective in introducing the development team to the technology they were to use.

What improvements in productivity resulted from the use of the Environment on this project? Using data from previous projects, SLOC per hour increased 118 percent, from 1.5 to 3.28 for the first ship. CelsiusTech projects an overall improvement of 627 percent, to 10.92, on ships 2-5. A natural question, however, is how much of the savings in manpower and time is due to use of the Environment, how much is due to reuse, and how much is due to use of Ada and object-oriented design? According to CelsiusTech, they saved $22.3 million on the first ship system, of which $14.6 million was attributed to use of the Environment and the remainder due to use of Ada and OOD. On the whole five-ship project they estimate a savings of $187.2 million, with $75.8 million due to reuse, $39.9 million to Ada and OOD, and $71.5 million to use of the Environment.

Computer Sciences Corporation and STANFINS-R[1]

Computer Sciences Corporation (CSC), founded in 1959, is a large, independent, professional services company. Their business is worldwide in scope, but a significant portion of it is with the U.S. Government.

The Standard Army Financial System (STANFINS), the U.S. Army's comprehensive computerized finance system, includes a broad range of accounting capabilities, including accounts receivable, funds receipt, general ledger,

[1] The primary sources for this section were (Rational 1991 Document CS-3), (Fussichen 1992), and (Puttré and Oppenheim 1989).

property accounting, and report generation. It was written in the 1960s using COBOL, and the STANFINS Redesign (STANFINS-R) contract, awarded to CSC in 1986, also specified COBOL. Although the Army issued a directive in late 1986 mandating use of Ada for all systems, CSC expected a waiver of this requirement. This expectation, coupled with the perception that Ada was appropriate only for embedded weapons systems, led CSC to spend five months preparing to use COBOL on this project. Doubting the effectiveness of Ada for the development of large-scale information systems and citing the lack of an Ada compiler and development tools for MVS, the Army director of finance and accounting requested a waiver on two separate occasions (Puttré and Oppenheim 1989). The waiver never came and in March 1987 the Army specifically directed CSC to use Ada for STANFINS-R.

STANFINS applications would run on top of IBM's Customer Information and Control System (CICS). CICS allows multiple users to concurrently access files, resolving issues related to multitasking, record locking, and terminal control (Fussichen 1992). CICS would in turn execute on IBM and Amdahl mainframes running the MVS operating system. CICS would also access the database management system specified in the contract, Datacom/DB. This combination of requirements, i.e., the use of Ada and the IBM mainframe target, jeopardized CSC's ability to deliver the system on time and within budget, Specifically, they were faced with the following serious problems.

- No major management information systems (MIS) application had been written in Ada in the U.S. at this time; therefore there was a lack of expertise both among CSC's MIS staff and in the MIS community in general.

- An Ada execution environment for the target platform was unavailable at the beginning of the project, so testing was initially impossible.

- There were no Ada interfaces to CICS, MVS, or Datacom/DB, so these would have to be written from scratch.

- There were no Ada-based CASE tools available for the target platform.

CSC took several steps to address these issues. First, they discussed execution-related problems with the compiler vendor (Intermetrics), who agreed to write a new run-time system to provide communication between Ada and CICS. Second, they decided to solve problems related to the development phase by acquiring the Environment. Reasoning that it would be easier to teach Ada to MIS programmers than MIS to Ada programmers, CSC began a recruiting and training program. Using Ada expertise from both inside and outside the company, CSC introduced its STANFINS team to the principles of software engineering and OOD, Ada syntax and semantics, and use of the Environment. With the assistance of consultants from Rational, CSC overcame the inability to test code on the mainframe by writing software scaffolds to run in the Environment which handled functions ultimately to be performed by CICS and Datacom/DB on the target. This idea worked so well that even when the host Ada environment became available, it was still preferable to test in the

Environment.

Another important issue was related to the size of the system. STANFINS-R was 2.4 million SLOC when delivered to the Army in May 1991; an additional 300,000 SLOC was produced for tool support. To deliver that much code and meet the contract deadline required production of more than one-half million Ada SLOC per year. This would not have been possible without automatic code generators. Again with help from Rational, CSC developed screen painters and other necessary tools which wrote Ada package specifications and bodies using information taken from Army MIS requirements. To perform their tasks correctly, these tools accessed a database containing data type definitions for more than 2500 data items. (Project management mantained strict control over this database and reserved all decisions regarding data typing to themselves; allowing the development staff the ability to change type descriptors would have resulted in chaos.) Ultimately 76 percent of the Ada code was produced by these tools (Fussichen 1992).

CSC believes that if they had naively substituted Ada for COBOL in their development process, there would have been no improvements in productivity. Using projections based on Ada industry norms, productivity on the STANFINS-R project would have been 1.8 SLOC per hour and the total labor costs would have been $48.2 million. Using the Environment, code was produced at at rate of 3.7 SLOC per hour and labor costs were $24 million. Finally, and perhaps most importantly, CSC met every delivery date specified in the contract.

IBM and WISCUC[1]

Completing a large application in a timely manner often requires several development teams using multiple hardware platforms. This situation immediately introduces problems with maintaining consistency and performing system tests. This case study describes how one group of developers solved these problems.

The World Wide Military Command and Control System (WWMCCS) is a large and complex assembly of hardware, software, and communications subsystems under development for the U.S. military. A critical component of this assembly was a generalized automated message handler (AMH) developed by IBM's Federal Systems Company under the WWMCCS Information System Common User Contract (WISCUC or WIS). This project began in 1983 and the AMH was delivered in 1990. The AMH target hardware included an IBM System/370 mainframe running MVS/XA and two Series/1 communications processors. Additionally, users access the AMH through WISCUC workstations based on the IBM 3270/PC. Under the WISCUC, software was developed for all three types of hardware, but the mainframe component discussed here consisted of

[1] The primary source for this section was (Blair 1992).

about 300,000 Ada SLOC.

Software development required five processors. All coding and unit testing, as well as part of the component and integration testing, were done on Rational R1000s; three of these systems were required to provide sufficient disk storage for the project. An MVS/XA Software Development Laboratory (SDL) running on an IBM 3081 was used for the rest of the component and integration testing. Finally, an MVS/XA Testing and Integration Laboratory running on the IBM 4381 target was used for system testing. This last step was accomplished by transferring the Ada source code from the Rationals over a TCP/IP link into Software Configuration Library Manager datasets on the target for compilation under the Intermetrics Ada Development System.

The AMH software structure included six major components: four Computer Program Configuration Items (CPCIs), global packages, and auxiliary code for integration testing. The CPCIs were further broken down into a total of sixteen Computer Program Components (CPCs); each of these was implemented as a Rational subsystem. Multiple spec and load views of each subsystem were required for implementation and testing. There were three types of load views. Code in the "released" load view could not be modified. The "master working" load view contained code still subject to change, but which would be promoted to the released view the next time a program baseline (snapshot) was created. Finally, there was at least one subsidiary working load view containing code still subject to significant modification.

The source code was distributed across three R1000s due to the disk space limitation noted earlier. Each R1000 owned a "primary" copy of two of the six major software components and "secondary" copies of the other four. A primary copy included spec, released, and working views of subsystems for that component, while a secondary copy included only spec and released views. Therefore developers on a particular system could modify code in only two of the six components. This distributed mode of operation required a carefully structured procedure, followed once a month, to create a new baseline version of the program.

- Incorporate necessary changes into the master working view using CMVC.
- Build new spec views, if necessary.
- Recompile the master working view.
- Copy the primary spec views to secondary spec views on other R1000s.
- Freeze the just-compiled master working view using CMVC and make it the new released view.
- Copy primary released views to secondary released views on other R1000s.

- Update all other working views using the new spec views.
- Update the system activity file to indicate which load view of each subsystem to use for subsequent executions.

Over the course of this project the IBM development staff made several significant observations regarding use of the Environment. First, their experience verified Rational's claim that the Rational compiler minimized the scope of recompilation. Second, use of CMVC in conjunction with separate activity files to specify different versions of the program allowed development and testing to proceed concurrently. Third, designation of a separate team for version control ensured that the previously described procedure was correctly followed and that development and testing would not periodically grind to a halt. Finally, lack of sufficient disk space on a single R1000 system was a significant disadvantage, but one which was accepted due to the other positive features provided by the Environment and to the lack of an acceptable Ada development environment on the target.

Statistica and SIDPERS-3[1]

The original version of the Army's Standard Installation/Division Personnel System (SIDPERS) was first deployed in 1973. The original purpose of the system was to assist in personnel management, keeping track of the duty status of all active Army personnel. SIDPERS-1, as it is now called, was written in COBOL and executed on IBM 360 Model 40 mainframes installed at the base and division level. Clerical personnel below those levels had to manually prepare and submit their reports for processing on those mainframes.

As Army requirements increased, additional capabilities were added to SIDPERS, including facilities to handle redeployment, casualty reporting, and promotions. As might be expected, the target hardware also evolved over time; Amdahl and Burroughs equipment was installed and automation was provided for lower levels in the organizational hierarchy. Although in many ways the system satisfied the Army's needs, there were problems. The various Army commands requested numerous modifications to the system and the SIDPERS maintenance group was unable to implement them in a timely manner. Unwilling to wait any longer, the commands authorized their own local customizations of the code. Standardization was lost and maintenance costs and personnel increased.

This situation led the Army to initiate development of what will become SIDPERS-3[2]. The overall goal of this system is to produce a standard system which may be easily extended to accommodate the unique needs of each

---

[1] The primary source for this section was (Statistica 1991).

[2] Yes, there was a SIDPERS-2; in fact there was even a SIDPERS-2.5 and a SIDPERS-2.75, but that, as they say, is another story.

command. Specific objectives to be met by SIDPERS-3 are to:

- Provide the capability to determine how many qualified personnel are needed at each Army site in order to enhance the Army's ability to mobilize and deploy its forces.
- Produce an easy-to-use system which is accessible to users at all levels.
- Provide a communications capability which allows reporting throughout the organizational hierarchy.
- Reduce the personnel required to manage the personnel system itself by automating manual processes and allowing interactive transactions against the personnel database.
- Provide a single source of personnel data, thereby reducing the possibility of redundant and inconsistent information.
- Improve response time.

Statistica is a professional services company headquartered in Rockville, Maryland. Founded in 1977 and with a 1992 employment of about 400, it is younger and smaller than many of the companies with which it must compete. Nevertheless, Statistica has been successful, having completed every one of its contracts within time and budgetary constraints. Some of the reasons for this success include the company's object-oriented approach to Ada software engineering and its willingness to use new software development technology, including prototyping, reuse, and CASE tools. Importantly, Statisitca has adopted the SEI Capability Maturity Model (CMM) (Paulk et al. 1993 Technical Report CMU/SEI-93-TR-24, Paulk et al. 1993 Technical Report CMU/SEI-93-TR-25) to measure the quality of its software developent process.

Statistica was selected as the lead contractor on the SIDPERS project; SRA, Martin Marietta, and Planning Analysis Corporation were subcontractors. Because the Army had not yet selected the target hardware and because there was a high liklihood of multiple targets, Statistica chose the Environment as its APSE. In addition to four R1000s, Statistica equipped its development facility with various systems then installed at Army bases: an IBM 4341 mainframe, a Sperry 5000, 68 Zenith-248 PCs, two TACCS-E systems, a Unisys B-38, and several Everex 386 PCs running UNIX. In such a heterogeneous environment, it was obviously in the company's interest to keep developers on the Rational platform as long as possible. This was facilitated by Rational consultants who assisted Statistica in the implementation of a remote procedure call (RPC) capability for the Environment. Thus, instead of merely simulating the database transactions, testing was able to proceed on the R1000s by accessing the actual DBMS on the Everex systems. The result was highly portable Ada code. The 60,000 SLOC demonstration system, roughly 10% of the final product, is operational on several platforms, and the man-machine interface has been installed on six different targets with negligible technical difficulties.

In their appraisal of the Environment, Statistica noted its support for modularization, the Ada-sensitive editor, incremental compilation, RDF support for requirements traceability, provision for host-based testing via RPCs, and the ease of porting to the target using the TBU. Statistica had particular praise for Rational's technical support. They further emphasized the power and sophistication of the environment; subsystems enabled them to break the system into manageable pieces having clean interfaces; incremental compilation reduced compilation time relative to other APSEs; RDF support for DOD-STD-2167A promoted code consistency between developers; use of RPCs allowed more debugging to take place on the Rational host, thus uncovering many problems prior to actual testing on the target.

Statistica was candid about some of the disadvantages to using the Environment. These included the high system cost, performance degradation with more than ten simultaneous users per R1000, the extensive training required to use the system effectively, the customization, or "toolsmithing," required to use the RDF, TBU, and RPCs, and finally the lack of a graphical design capability. Nevertheless, Statistica believes that Rational was the best choice, concluding that other APSEs have similar problems, and provide far less functionality than the Environment.

The Software Engineering Institute Evaluation[1]

A primary goal of Carnegie Mellon University's Software Engineering Institute (SEI), is to critically evaluate state-of-the-art technology for software development and disseminate the results. As one means to reach this goal, the SEI initiated its Evaluation of Ada Environments (EAE) Project in 1985 with the objective of assessing the capabilities of APSEs. The first principal outcome of this project was the specification of a methodology for evaluating APSEs (Weiderman et al. 1986). The second major result was the application of this methodology in a comparative study of three APSEs: the Ada Language System, Verdix Ada, and DEC Ada (Weiderman et al. 1987). The SEI's efforts in this area were continued under the Evaluation of Environments Project. Three major results of this project have been the extension of the EAE methodology to include SDEs and IPSEs, an evaluation of the ISTAR IPSE (Graham and Miller 1988), and an evaluation of the Environment.

The SEI assessment of the Environment was conducted in 1988. The hardware component of the evalauted system consisted of an R1000 Model 200-20 with 32 Mbytes of main memory, three disk drives each with an unformatted capacity of about 670 Mbytes, a 75 ips streaming tape drive, an Ethernet interface, and eight Rational terminals. The system software was Release D_9_25_1 of the Environment; it included the basic operating system, a tiled window system for the terminals, language-sensitive editor, compiler, debugger,

[1] The primary sources for this section were (Feiler, Dart, and Downey 1988) and (Downey, Bassman, and Dahlke 1988).

CMVC, and support for workorder management. Layered products, such as the RDF, TBU, and Rational Network Mail, were not included in the evalaution due to their unavailability. The evaluation process included both the application of the SEI methodology and additional analysis of unique features of the Environment not addressed by the methodology.

The SEI evaluation methodology includes six categories of experiments, five of which were used in the Rational study. These experiments explore in a detailed manner the capabilities of an APSE and typically require extensive hands-on use of the system.[1] The CMVS category includes three experiments, the first of which models the system integration and testing phases of a software development project. The second experiment builds on the results of the first; it involves creation of a model software system, followed by construction of several baseline versions representing various stages of progress in the development process. The environment is then tested to determine its ability to reconstruct previous versions and to build a composite version using components from current and previous versions. The functionality checklists for these two experiments (Downey, Bassman, and Dahlke 1988, p. 25-26, 39) indicate that the environment supported every primary activity in all three areas of interest: version control, configuration control, and product release.

The last experiment in the CMVS category evaluates an APSE's software management policy. The Environment allows project management to specify its own policy. Because this experiment was designed to examine an existing policy and does not specify a candidate policy, it was omitted from the evaluation.

The second category contains system management experiments which address APSE installation, management of user accounts, and system usage accounting support. A few of the capabilities required by these experiments were not directly provided by the Environment, but were provided in other ways, such as writing a special-purpose procedure. This may have been due to differences in SEI experiences with previously tested environments and Rational's philosophy of system architecture. In any case, these experiments indicated few shortcomings in the Environment's system management capabilities. Further details of this experiment may be found in (Downey, Bassman, and Dahlke 1988, p. 51-73)

Design and development capabilities are assessed by the third category. Part of this section was omitted because the Environment lacked (at that time) graphical design tools. This experiment involves entering an Ada program with known errors and then evaluating an environment's ability to detect them. The Environment provided all but two of the 27 capabilities requested (Downey, Bassman, and Dahlke 1988, p. 94-95).

---

[1] The SEI technical report (Downey, Bassman, and Dahlke 1988), which contains the transcripts of the experiments, is 185 pages long; its length reflects the thoroughness of the evaluation.

The fourth category consists of a unit testing and debugging experiment. It evaluates an APSE's capabilities for code browsing, debugging, regression testing, and static and dynamic code analysis. At the time these tests were conducted, the Environment had no tools for these latter types of analyses. The Environment debugger fared rather well in this test, and it is important to note that some of the requested functionality has since been provided by the Testmate product. See (Downey, Bassman, and Dahlke 1988, p. 115-139) for more detailed information on the results of this experiment.

The fifth section assesses the project management capabilities of an APSE in four areas: project plan management, plan instantiation, project execution, and product management. Because the Environment's capabilities in this area had already been tested in the CMVC experiments, this category was omitted from the evaluation.

The final component of the SEI evaluation methodology is the execution of the Ada Compiler Evaluation Capability (ACEC) test suite (Hook et al. 1985), which was then in a prototype version.[1] On this test, the Environment compiler was 1.8 times faster than the VMS/VAXSet compiler and 2.1 times faster than the UNIX/VADS compiler; the Environment was also somewhat faster than these two systems in program execution time. More detailed results from this experiment may be found in (Downey, Bassman, and Dahlke 1988, p. 141-153), but the reader is cautioned that these performance data are no longer representative of the performance of the various environments.

SEI reached several conclusions regarding the Environment. First, although their results offer a basis for comparison between the Environment and other systems, they cautioned that the standard SEI evaluation methodology did not completely characterize the full functionality of the Environment. Next, they concluded that the Environment "provides a powerful and effective semantics-based interaction model."[2] Users are thus able to browse Ada code based on its syntax and semantics (e.g., intermodule dependencies) and to obtain semantic information about the code (e.g., the amount of recompilation required by a particular code modification). Under this model, syntax and semantic errors may be detected and corrected incrementally through frequent invocation, from the editor, of the parser and semantic analyzer. The use of this intelligent, incremental compilation feature, coupled with the dynamic linking capabilities of the Environment has the potential to reduce the amount of recompilation required after a change.

---

[1] Refer to (Wright Research and Development Center 1988 Technical Report AFWAL-TR-88-1095) for a more up-to-date description of the ACEC. In July 1993 AJPO announced completion of the merger of the ACEC with its United Kingdom counterpart, the Ada Evaluation System (AES) (Ada Information Clearinghouse 1993). The new test suite is called the Ada Compiler Evaluation System (ACES).

[2] From (Feiler, Dart, and Downey 1988, p. 71); © 1988 SEI, reprinted by permission.

The use of DIANA as the underlying program representation was identified as the enabling factor in the integration of development and debugging facilities. Moreover, effective use of the system does not require knowledge of DIANA. The Environment encourages early testing and rapid protyping via mechanisms for generation of code for stubs and for incomplete modules. Rational's subsystem concept supports development of large systems by providing a level of program modularity above the package; this feature helps limit the scope of a recompilation.

Although certain activities, such as a large-scale recompilation or large system test may seriously degrade response time, SEI found that the Environment, as a system, was comparable to other widely-used APSEs in terms of compilation speed and disk utilization. The Environment was described as providing "a highly responsive system by deploying smart compilation and dynamic linking techniques and through cooperative input from the user to reduce reprocessing after changes based on semantic information."[1] An important final comment noted that because the Environment does not support all phases of the software life cycle, other tools must be acquired and integrated with it to provide a complete software development environment.

Magnavox Electronic Systems Company and AFATDS[2]

In early 1990, Magnavox Electronic Systems Company (MESC) conducted an evaluation of three candidate development software support environments (DSSEs[3]) for use on the Advanced Field Artillery Tactical Data System (AFATDS) project. The DSSE provided by the army was the Army Tactical Command and Control System (ATCCS) programming support environment (PSE); the evaluation discussed here was motivated by the discovery of serious deficiencies in that environment. MESC included the DEC VAX environment in its study because they had used that system in the concept evaluation phase (CEP) of AFATDS. In spite of its shortcomings, the ATCCS was included in the evaluation to serve as a baseline. Finally, at the suggestion of another contractor, the Environment was added as a candidate. The actual configurations evaluated by MESC are given in Table 1.

The multi-element component comparison and analysis (MECCA) methodology (Ulvila and Brown 1982) was used in the MESC evaluation. Application of the MECCA technique requires development of a hierarchy of weighted evaluation attributes. This hierarchy may be viewed as a tree, each node of which contains a weight and a score. For this method to work properly, each parent attribute must be partitioned into independent subattributes so that some features are

---

[1] From (Feiler, Dart, and Downey 1988, p. 76); © 1988 SEI, reprinted by permission.

[2] The primary source for this section was (Magnavox Electronic Systems Company 1990).

[3] A DSSE consists of an APSE and the associated host hardware.

Table 1
MESC Ada DSSE Tools Evaluation[1]

Feature	Candidate DSSE		
	DEC VAX	Rational R1000	ATCCS PSE
CPU(s)	VAX 8600, 8650, 8800	R1000 Model 200-10	HP 9000/350
Operating system	VMS v. 5.2[2]	Rational Environment v. D_10_20_0[2]	HP-UX v. 6.2[2]
Project mgmt. tools	VAX Software Project Manager v. 1.2[8]	Rational Work Order[2]	
Design tools	Cadre Teamwork[8] IDE STP[8]	Rational RDF v. 6_2_5[2] RDF Teamwork Interface v. 1_0_0[2]	Cadre Teamwork[8] IDE STP[8] Mark V Systems Adagen
Text editors	VAXTPU EVE v. 2.0[2] DEC LSE v. 2.2[3]	Rational Editor[2]	HP-UX vi[2] emacs[8]
Document support	DEC Runoff[3]	Rational Document Formatter v. 10_7_7[4] Rational Interleaf Interface[8]	HP-UX nroff[8]
GUI support	DECWindows[8]	Rational RXI v. β[2]	X v. 11.3[2]
Ada compiler	DEC Ada v. 2.0[2]	Rational Compiler[2]	HP (Alsys) Ada v. 3.25[2]
Ada library & recompilation support	DEC Ada v. 2.0[2]	Rational Library Manager[2]	GEC G-ADA /HP Ada 300 ada.make v. 2.0[2]
Ada linker	DEC Ada v. 1.5[3]	Rational Linker[2]	HP Ada v. 3.25[2]
Ada debugger	DEC Debugger v. 5.0[3]	Rational Debugger[2]	HP Ada ada.probe v. 3.25[7]
Static analyzers	DEC SCA v. 2.0[8] EVB Software CMT[3] DRC AdaMAT[3]	Rational Xref v. 9_1_2[2] EVB Software CMT[5] DRC AdaMAT[5]	HP Ada ada.xref v. 3.25[2]
Dynamic analyzers	DEC PCA v. 1.1[3]	Rational Performance Analysis Interface[5]	
Host-target integration tools		Rational MC68020 Bare CDF v. 5_1_0[2] Rational HP-UX CDF[5] Rational TBU v. 9_4_4[2]	
Sys. test tools	DEC DTM v. 3.0[8]		
Configuration management support	DEC CMS v. 3.0[3] Expertware CMF[7] Softool CCC[6] Magnavox MACE[3]	Rational CMVC[2]	HP-UX SCCS[7] HP-UX RCS[7] Expertware CMF[7] Softool CCC[6]

[1] From (Magnavox Electronic Systems Company 1990, p. 4)
[2] Evaluation based on experiments during DSSE evaluation
[3] Evaluation based on experience during AFATDS CEP
[4] Evaluation based on demonstration by vendor
[5] Evaluation based on information from vendor
[6] Evaluation based on information from other projects or customers
[7] Evaluation based on product documentation
[8] Not Evaluated—tool was unavailable, or time constraints prevented evaluation

not scored more than once. The sum of the weights of the child nodes of any particular parent must be one. Scores for an attribute must fall between 0 (''useless'') and 100 (''ideal''). The scoring process begins by assigning scores to each of the leaf nodes in the tree. A score is calculated for each parent by summing the weighted values of its children. This continues until the overall score associated with the root of the tree is computed.

MESC used four attributes at the highest level of the MECCA tree. The first of these, ''system management,'' was weighted 14% and scored a DSSE in the areas of system administration, training, vendor responsiveness, and system performance. ''Project management,'' the second attribute, was weighted 20% and addressed administrative issues related to the software project itself (e.g., scheduling, library and file system functionality, and support for quality management). ''Technical development'' counted 40%, the largest weight of any of the four primary attributes, because more manpower would be associated with activities in this area than any other. This category evaluated a DSSE's support for software engineering, including analysis, design, implementation, testing, and documentation. The last category, weighted 26%, was ''configuration management;'' it addresses issues related to version control, change control, and building and rebuilding the system.

MESC took two important steps to reduce subjectivity in the scoring process. First, a separate worksheet was used to score each low level attribute. On this worksheet each evaluator described his findings (e.g., experimental results, vendor information, previous customer experience) reported his analysis of these findings in a conclusions section, and then finally assigned a score for that attribute. This standardization of reporting also made this summarization process easier.

Second, several techniques were instituted to standardize the scoring process and reduce bias, including:

- Assigning each DSSE a base score of 50 and then adding points to reflect the level of automation provided by that environment for a particular attribute.

- Assigning additional points (typically 10) for experience with an environment during the CEP. This technique was used to reflect reduced risk with respect to a particular attribute. Obviously this improved the score of the VAX/VMS DSSE.

- Assigning a score of 100 and then deducting points for deficiencies.

Occasionally other techniques were used, but all were applied consistently; scoring strategies were always uniformly applied for all DSSEs in scoring a particular attribute.

The results of the MESC evaluation for the primary attributes are given in Table 2. Although all three candidate environments were quite close in the scoring for system management, VAX/VMS had a slight edge due to the experience

Table 2
MESC Ada DSSE Evaluation Scores[1]

Attribute	Weight	Candidate DSSE		
		DEC VAX	Rational R1000	ATCCS PSE
System Management	0.14	93.78	91.82	89.52
Project Management	0.20	69.91	79.08	59.75
Technical Development	0.40	58.42	68.16	49.69
Configuration Management	0.26	93.93	92.50	87.93
Total	**1.00**	**74.90**	**79.98**	**67.22**

[1] Adapted from (Magnavox Electronic Systems Company 1990, p. 30-31)

with that system during the AFATDS CEP. The Environment was clearly superior to the other two DSSEs in the project management category. The reasons for this were as follows. First, the Rational work order facility permitted tracking of project tasks by allowing allowing management to associate a work order with the modification or correction of a particular Ada unit.

Second, the Environment outscored the other systems in the consistency checks/alerts category because it notifies a developer of the impact of a proposed change. When a change would affect many units and require significant recompilation, then the developer could elect to delay the modification until the next major system build. Finally, Rational was the winner in the project reviews and walkthroughs section because it allows managers to specify their own coding standards and then enforces them automatically. No recompilation is required by this feature because syntactic and semantic differences, not time stamps, determine a unit's consistency.

The evaluation for the technical development category was performed by transporting the AFATDS CEP Human Interface (HI) software, developed on the VAX, to the ATCCS PSE and to the R1000. This allowed quantitative evaluations of resource requirements and provided hands-on experience with each of these systems. The HI code metrics are given in Table 3.
Resource consumption figures are presented in Table 4. "Parsing" in this context includes scanning the source code, checking syntax, and entering dependency information into an Ada library. Although the R1000 parse time is high, MESC noted that parsing an entire system would be a rare occurrence (presumably due to the R1000's persistent intermediate representation of the program in DIANA). In evaluating R1000 disk space utilization, MESC noted that copies of lower-level supporting code would have to be present on each R1000, a situation also encountered by IBM (Blair 1992).

As part of the HI porting experiment, MESC evaluated the various capabilities of the candidate systems. Some of the features of the R1000 which

Table 3
Code Metrics of HI Source Files[1]

Metric	Number
Comment lines	70,500
Blank lines	53,000
Non-comment, non-blank lines	90,000
Non-literal semicolons	43,000
Total source lines	**213,500**

[1] From (Magnavox Electronic Systems Company 1990, p. 17)

Table 4
Performance Comparison for HI Port[1]

Metric	Candidate DSSE		
	DEC VAX	Rational R1000	ATCCS PSE
"Parsing" time (hours:minutes)	0:26	3:00	0:30[2]
Compile time (hours:minutes)	2:24	2:20	15:23
Library disk space (Kbytes)	19,967	33,608	55,071

[1] From (Magnavox Electronic Systems Company 1990, p. 18)
[2] Time estimated; parsing failed for 2 files (out of about 700).

distinguished it from the other DSSEs included integration of the editor with the pretty printer, ability to semantically browse the code, ability to specify and automatically enforce coding standards, the subsystem concept, and ability to build composite versions of a system from different subsystem components. A summary of this feature comparison is given in Table 5.

The final primary attribute was configuration management. MESC noted that Rational's CMVC was superior with respect to the integration subattribute due to its capabilities for building and rebuilding systems. Rational's score was reduced by 10 points in each of five subattributes where additional software was required to augment CMVC. In spite of this, VAX/VMS beat the R1000 in this category by less than 2 points.

The MESC study recommended that the Environment, augmented with components from the ATCCS PSE, be selected as the AFATDS DSSE. The reasons for this were as follows:

- Rational had the highest overall score (Rational 79.98%, VAX/VMS 74.90%, and ATCCS PSE 67.22%).
- Rational is an Ada language-centered APSE with capabilities unavailable in other environments.

Table 5
Summary of Technical Development Features[1]

DSSE Tool	Tool Feature	Candidate DSSE		
		DEC VAX	Rational R1000	ATCCS PSE
Editor	Language sensitive	X	X	
	Integrated with pretty printer		X	X[2]
	Traverses Ada semantic network		X	
Compiler	Enforces design, coding standards		X	
	Presents obsolence report		X	X[3]
	Recognizes object dependencies		X	
	Integrated with document builder		X	
Linker	Runtime alternate implementations		X	
Library manager	Automated compile order	X	X	X[4]
	Automated library recompile	X	X	X[5]
	Sublibraries	X		
	Subsystems		X	
Debugger	Source level debugging	X	X	X
	Ada tasking support	X	X	
System integration	Configure by source files	X	X	X
	Configure by subsystem		X	
	Target build functionality		X	NA[6]

[1] From (Magnavox Electronic Systems Company 1990, p. 25)

[2] The HP Ada 3.25 pretty printer ada.format is a separate tool.

[3] The HP Ada 3.25 Ada library manager ada.umgr does not detect dependencies across multiple program libraries.

[4] This feature is not provided by HP Ada 3.25, but by the G-ADA ada.depend tool.

[5] This feature is not provided by HP Ada 3.25, but by the G-ADA ada.make tool.

[6] This feature is not needed for the PSE, since the development and target environments are the same.

- Rational promotes the development of high quality software by supporting recognized software engineering principles auch as abstraction and information hiding.

- Rational, augmented by the ATCCS PSE, will provide a scalable software development environment capable of attacking large-scale programming projects.

- Rational's TBU and RPC provide an integrated host-target environment which smooths the transition from development on the host to installation and testing on the target.

- Rational allows custom specification and automatic enforcement of coding standards.

- Rational's editor and compiler provide for automated collection of software metrics and generation of metrics reports.

The concluding statements in the MESC report were unequivocal:

> The existing Rational system management, project management, technical development, and configuartion management tools may be integrated with additional software to create an environment to encompass the total software development process. When software developers, managers, testers, or other personnel log into this augmented Rational system, they will be logging into a configuration management environment and will perform all functions within this environment. Neither alternative provides the possibility of such an environment in the forseeable future (Magnavox Electronic Systems Company 1990, p. 33).

Other Users

Honeywell announced in January 1994 that it will use Rational Apex on Sun SPARC workstations to develop the Boeing 777 Airplane Information Management System (AIMS); AIMS is an avionics software application which controls both the 777's flight management computer system and its central maintenance controls.

France's national railroad organization, Societe Nationale Chemins de fer Francais (SNCF), has selected the Environment as the development platform for its Astree project. SNCF plans to equip every train with on-board systems to transmit its location, speed, and other operational information to DEC VAX/VMS systems in Astree processing centers. These systems will monitor train position and performance to ensure passenger safety as well as smooth operation.

Other documented examples of projects using the Environment include the Canadian Automated Air Traffic System (CAATS) under development by Hughes Aircraft of Canada (The Rational Watch Winter 1992), IBM's Advanced Automation System (AAS) contract with the Federal Aviation Administration (Taft 1990), development of the NASA space station data management system by Lockheed (Suydam 1991), and TRW's Universal Network Architecture Services (UNAS) product (Crafts 1993, Royce et al. 1991).

5 Recommendations and Conclusions

Recommendations

Based on the information gathered in this study, the author gives an unqualified positive recommendation for acquisition of the Rational Environment if it is to be used on large (100,000+ SLOC) projects. This positive recommendation can be further extended to the situation where multiple moderate size projects are contemplated. If only small systems will be developed, the Environment will still be quite helpful in the application sense, but it will probably not be cost effective (although Rational Apex pricing may alter this conclusion). If the Environment is acquired, the following guidelines should be followed to ensure its successful utilization.

- Adopt a strategic perspective with respect to the software development process. A good starting point for guidance in this respect is (Hefley et al. 1992).
- Adopt the SEI CMM as a means of improving the organization's development process; plan to reach CMM Level 2 (''repeatable'') within a year.
- Do not view the Environment as a ''silver bullet;'' (Brooks 1987) decide on a realistic development methodology and use the Environment as a tool to implement that methodology.
- Acquire appropriate layered products, e.g., Insight and Testmate, to cover the various phases of the software life cycle specified by the methodology.
- Plan to use the RXI as the means of accessing the Environment; users will find it more functional than the Rational Windows Interface (RWI).
- If there are sufficient funds, equip each developer with a UNIX workstation having a large monitor. If cost is an issue, acquire X terminals or PC-based X terminal emulators to run the RXI.

- Plan to migrate to Rational Apex when the layered tools become available, probably in late 1994.
- Acquire the services of Rational consultants for some initial period.
- Use these consultants to train a core group to serve as in-house experts.
- Build further expertise by using the Environment on a small pilot project (Brooks 1975, p. 116).
- Use the experience gained in the pilot to identify necessary changes in methodology and additional necessary tools.
- Provide formal training on a "just-in-time" basis for additional developers. As part of this training require every developer to read *The Mythical Man Month* (Brooks 1975); it is an essential reference for the entire staff.
- Assign a member of the core group to each development team to serve as a mentor.
- Assign one or more individuals to handle special tasks, e.g., configuration management, testing, and toolsmithing (Brooks 1975, p. 128).
- Document the way the Environment was used to implement the methodology so that future projects can build on newly acquired knowledge.

Conclusions

The basic Environment provides powerful aids for software development in Ada, particularly in the areas of coding, browsing, configuration management, and version control. The subsystem concept, which provides a level of abstraction above the package, is crucial to the development of very large systems. The underlying, DIANA-based intermediate representation is the basis for many of the environment's important features, e.g., syntactic completion and semantic verification from within the Ada-knowledgeable editor. The DIANA representation, along with the object-oriented multistate approach to code translation, helps limit the scope of a recompilation. Again, this is particularly useful for large projects.

The Environment is a back-end CASE tool environment; it is critical that it be augmented by one or more other tools which address the requirements analysis and design phases. Examples of such tools include the RDF, Insight, and Teamwork. The RDF, which uses a PDL-based approach to system design, is worthy of particular mention. Because this PDL is a subset of Ada, many design tasks can be automated, interfaces and other relationships between design elements may be automatically verified, and the PDL design itself can naturally evolve into the Ada implementation. Either Insight or Teamwork may be used as a graphical front-end to the RDF.

There are tools which address other aspects of the development process. One of the more significant is Rational Testmate, which automates many aspects of unit, integration, and regression testing. Testmate manages the complexity associated with developing, running, and maintaining the large numbers of test sets required to validate multiple versions of a large software system. Another important tool is DRC's Adamat, which gathers data about the actual Ada code and uses that information to compute various software metrics. This allows measurement and improvement of software quality. Unfortunately, an important component which has not yet been provided is an Environment tool which addresses business process modeling.

One of the strongest criticisms of the Environment has been the expensive proprietary hardware and operating system required to run it. Rational Apex, an implementation of the environment for RISC-based UNIX workstations introduced in August 1993, answers that criticsm. More than just a direct ''port'' of the environment, Apex takes advantage of underlying UNIX features, e.g., NFS and file ownership and security mechanisms. Furthermore, the Apex interface is a Motif-compliant application running under the X Window System. Rational's supporting tools, e.g., Testmate and Insight, will be available for Apex in 1994.

A development which could have a favorable influence on the Environment and its capabilities is the recently announced merger between Rational and the Verdix Corporation (Alexander 1993). Until this merger, the Verdix Ada Development System (VADS) APSE (Matthews and Burns 1991) probably presented the toughest competition to the Environment in terms of market share and technical merit. The VADS APSE is highly functional and it is reasonable to assume that many of its best features will ultimately be incorporated into the Environment.

Perhaps the most important advantage of the Environment is the manner in which it facilitates the development of large, complex Ada systems. Ada itself was designed to support such large system development and experience has shown it does so in ways no other language can (Deputy Assistant Secretary of the Air Force for Communications, Computers, and Logistics 1991). But even so, such projects can overwhelm a development effort when they reach several hundred thousand SLOC; the Environment allows developers to break through this barrier. Furthermore, even for projects of moderate size (10,000-100,000 SLOC), the improvements in developer productivity make acquisition of the environment an investment rather than an expenditure. The information gathered in the course of work, which includes interviews with environment users, leads to the conclusion that the Rational Environment is preeminent among Ada development environments, and arguably also among those available for any language.

References

Ada Information Clearinghouse. (May 1993). "Merger of performance-test suites completed," *Ada Information Clearinghouse Newsletter* 11(1), 7, 6.

Alexander, R. (Winter 1993). "Q&A: Ralph Alexander, president of Verdix," *The Rational Watch* 3(4), 6-7.

Amos, C. (Summer 1993). "Q&A: Carol Amos, marketing manager, on Rational Apex," *The Rational Watch* 3(2), 6-8.

Bachman, B. and Marasco, J. (July 1992). "Project results: benefits of OOD on large systems," Document CS-4, Rational, Santa Clara, California.

Backus, J. (August 1978). "Can programming be liberated from the von Neumann style? a functional style and its algebra of programs," *Communications of the ACM* 21(8), 613-641.

Barstow, D. R., Shrobe, H. E., and Sandewall, E. ed. (1984). *Interactive programming environments*. McGraw-Hill, New York.

Bennett, K. H. ed. (1989). *Software engineering environments: research and practice*. Ellis Horwood, Chichester.

Blair, D. J. (November 1992). "Managing Ada using Rational's Configuration Management/Version Control and IBM's Software Configuration Library Manager." *TRI-Ada '92 Proceedings*. C. B. Engle, Jr., ed., Association for Computing Machinery, New York, 424-431.

Booch, G. (1987). *Software engineering with Ada*. Benjamin/Cummings, Menlo Park, California.

Brooks, F. P. (1975). *The mythical man-month: essays on software engineering*. Addison-Wesley, Reading, Massachusetts.

_____. (April 1987). "No silver bullet: essence and accidents of software engineering," *Computer* 20(4), 10-19.

Penedo, M. H. (May 1992). "An annotated bibliography on integration in software engineering environments," Special Report CMU/SEI-92-SR-8, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pensylvania.

Brown, A. W., Earl, A. N., and McDermid, J. A. (1992). *Software engineering environments: automated support for software engineering*. The McGraw-Hill international series in software engineering, McGraw-Hill, London.

Buhr, R. J. A. (1984). *System design with Ada*. Prentice-Hall, Englewood Cliffs, New Jersey.

Buxton, J. N. and Druffel, L. E. (October 1980). "Rationale for STONEMAN." *Fourth International Computer Software and Applications Conference*. IEEE Computer Society, 66-72.

Byrne, W. E. (1991). *Software design techniques for large Ada systems*. Digital Press, Bedford, Massachusetts.

Cadre. (March 1991). "Integration of Teamwork/Ada with Rational," Application Note C-ANR.

_____. (December 1991). "Teamwork DSE," Publication C-PBDSE.

_____. (March 1991). "Teamwork SA," Publication C-PBSA.

_____. (June 1991). "Teamwork Ada," Publication C-PBTADA.

Caruso, D. (8 July 1985). "Development system breaks productivity barrier," *Electronics* 58(27), 36-40.

Comptroller General of the United States. (9 November 1979). "Contracting for computer software development--serious problems require management attention to avoid wasting additional millions," Report FGMSD-80-4, United States General Accounting Office, Washington, D.C..

Crafts, R. E. (June 1993). "Megaprogramming in practice-TRW's UNAS," *Ada Strategies* 7(6), 1-9.

D. Appleton Company. (February 1992). *Corporate information management process improvement methodology for DoD functional managers*, Fairfax, Virginia.

Dart, S. A. (December 1990). "Spectrum of functionality in configuration management systems," Technical Report CMU/SEI-90-TR-11, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pensylvania.

_____. (July 1992). "The past, present, and future of configuration management," Technical Report CMU/SEI-92-TR-8, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pensylvania.

Dawes, J., Pickett, M. J., and Wearing, A. (1990). *Selecting an Ada compilation system*. Ada Companion Series, Cambridge University Press, Cambridge.

Deputy Assistant Secretary of the Air Force for Communications, Computers, and Logistics. (July 1991). *Ada and C++: a business case analysis*. U.S. Department of the Air Force.

Deutsch, L. P. (1985). "Project support in the Smalltalk-80 integrated environment." *Integrated project support environments*. J. McDermid, ed., Peter Peregrinus, London, 124-134.

Devlin, M. T. (1980). "Introducing Ada: problems and potentials," unpublished report, USAF Satellite Control Facility.

Dolotta, T. A., Haight, R. C., and Mashey, J. R. (July-August 1978). "UNIX time-sharing system: the Programmer's Workbench," *The Bell System Technical Journal* 57(6, Part 2).

Downey, G. F., Bassman, M., and Dahlke, C. (September 1988). "Experient transcripts for the evaluation of the Rational Environment," Technical Report CMU/SEI-88-TR-21, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pensylvania.

Dynamics Research Corporation Systems Division. (1992). *Adamat Ada measurement and analysis tool user manual for the Rational*. Dynamics Research Corporation, Andover, Massachusetts.

Feiler, P. H. and Smeaton, R. (May 1988). "Managing development of very large systems: implications for integrated environment architectures," Technical Report CMU/SEI-88-TR-11, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pensylvania.

Feiler, P. H., Dart, S. A., and Downey, G. F. (September 1988). "Evaluation of the Rational Environment," Technical Report CMU/SEI-88-TR-15, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pensylvania.

Firth, R., Mosley, V., Pethia, R., Gold, L. R., and Wood, W. (1987). "A guide to the classification and assessment of software engineering tools," Technical Report CMU/SEI-87-TR-10, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pensylvania.

Fisher, D. A. (June 1976). "A common programming language for the Department of Defense --background and technical requirements," Report P-1191, Institute for Defense Analyses.

Frakes, W. B., Fox, C. J., and Nejmeh, B. A. (1991). *Software engineering in the UNIX/C environment*. Prentice Hall, Englewood Cliffs, New Jersey.

Fuggetta, A. (December 1993). "A classification of CASE technology," *Computer* 26(12), 25-38.

Fussichen, K. (November 1992). "Ada and CICS or (yes! Ada can be done on an IBM mainframe)." *TRI-Ada '92 Proceedings*. C. B. Engle, Jr., ed., Association for Computing Machinery, New York, 415-422.

Gibbs, N. and Ford, G. (1986). "The challenges of educating the next generation of software engineers," Technical Memo CMU/SEI-86-TM-7, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pensylvania.

Goldberg, A. and Robson, D. (1983). *Smalltalk-80: the language and its implementation*. Addison-Wesley, Reading, Massachusetts.

Goos, G., Wulf, W. A., Evans, A. Jr., and Butler, K. J. (1983). *DIANA: an intermediate language for Ada*. Lecture Notes in Computer Science, Springer-Verlag, Berlin.

Graham, M. H. and Miller, D. H. (August 1988). "ISTAR evaluation," Technical Report CMU/SEI-88-TR-3, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pensylvania.

Hefley, W. E., Foreman, J. T., Engle, C. B. Jr., and Goodenough, J. (October 1992). "Ada adoption handbook: a program manager's guide version 2.0," Technical Report CMU/SEI-92-TR-29, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pensylvania.

High Order Language Working Group. (July 1978). *DoD requirements for the programming environment for the common high order language, PEBBLEMAN*. U.S. Department of Defense.

_____. (January 1979). *DoD requirements for the programming environment for the common high order language, PEBBLEMAN revised*. U.S. Department of Defense.

_____. (February 1980). *DoD requirements for Ada programming support environments, STONEMAN*. U.S. Department of Defense.

Hitchon, C., Judd, M., Pritchett, G., and Thall, R. (30 September 1989). *Introduction to CAIS: common Ada programming support environment (APSE) interface set (MIL-STD-1838A)*. U.S. Department of Defense.

Hook, A. A., Riccardi, G. A., Vilot, M., and Welke, S. (October 1985). *User's manual for the prototype Ada Compiler Evaluation Capability (ACEC) version 1*. Institute for Defense Analyses, Alexandria, Virginia.

Hünke, H. ed. (1981). *Software engineering environments*. North-Holland, Amsterdam.

Kernighan, B. W. and Mashey, J. R. (April 1981). "The UNIX programming environment," *Computer* 14(4), 25-34.

Krasner, G. (1983). *Smalltalk-80: bits of history, words of advice*. Addison-Wesley, Reading, Massachusetts.

Lamson, M. (16 July 1991). *Rational Performance Model*. IBM Federal Sector Division, Boulder, Colorado.

Levine, S., Anderson, J. D., and Perkins, J. A. (March 1990). "Experience using automated metric frameworks in the review of Ada Source for AFATDS." *Proceedings of the 8th Annual National Conference on Ada Technology*. U.S. Army Communications-Electronics Command, Fort Monmouth, New Jersey, 597-612.

Long, F. ed. (1990). *Software engineering environments: international workshop on environments*. Lecture Notes in Computer Science, Springer-Verlag, Berlin.

Lyons, T. G. and Nissen, J. C. (1986). *Selecting an Ada environment*. Ada Companion Series, Cambridge University Press, Cambridge.

Magnavox Electronic Systems Company. (6 April 1990). *Development software support environment (DSSE) evaluation for Version 1 of the Advanced Field Artillery Tactical Data System (AFATDS)*, Fort Wayne, Indiana.

Matthews, E. and Burns, G. (Spring 1991). "VADS APSE: an integrated Ada programming support environment," *Ada Letters* 11(3), 61-72.

Mitze, R. W. (1989). "The UNIX system as a software engineering environment." *Software engineering environments: research and practice*. K. H. Bennett, ed., Ellis Horwood, Chichester, 345-358.

Morgan, T. M. (1988). "Configuration management and version control in the Rational Programming Environment." *Ada in Industry: Proceedings of the 1988 Ada-Europe International Conference*. S. Heilbrunner, ed., Ada Companion Series, Cambridge University Press, Cambridge.

Morris, E., Feiler, P., and Smith, D. (December 1991). "Case studies in environment integration," Technical Report CMU/SEI-91-TR-13, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pensylvania.

Newport, J. P. Jr. (28 April 1986). "A growing gap in software," *Fortune*.

Paulk, M. C., Curtis, B., Chrissis, M. B., and Weber, C. V. (February 1993). "Capability Maturity Model for software, Version 1.1," Technical Report CMU/SEI-93-TR-24, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pensylvania.

Paulk, M. C., Weber, C. V., Garcia, S. M., Chrissis, M. B., and Bush, M. (February 1993). "Key practices of the Capability Maturity Model, Version 1.1," Technical Report CMU/SEI-93-TR-25, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pensylvania.

Puttré, M. and Oppenheim, J. (27 February 1989). "Army rearms with Ada," *InformationWEEK*, 18.

Rational. (November 1986). "Large-system development and Rational Subsystems," Document 6004, Santa Clara, California.

_____. (20 May 1988). "Application of the Rational Environment to lifecycle software development," Document TO-1, Santa Clara, California.

_____. (July 1987). "Rational Environment basic operations," Product Number 4000-001116, Santa Clara, California.

_____. (November 1987). "Rational Environment user's guide," Product Number 4000-00117, Santa Clara, California.

_____. (August 1988). "Rational Environment reference manual volume 11: project management," Product Number 4000-00129, Santa Clara, California.

_____. (1990). "Rational design facility: Rational Teamwork interface," Product Number 4000-00602, Santa Clara, California.

_____. (19??). "Publishing interface: Interleaf TPS," Product Number 4000-00334, Santa Clara, California.

_____. (September 1989). "Rational design facility: DOD-STD-2167A user's manual," Product Number 4000-00362, Santa Clara, California.

_____. (August 1991). "Case study: foundation for competitiveness and profitability: FS 2000 System, Rational, and Ada," Document CS-1, Santa Clara, California.

_____. (June 1991). "Case study: automated tools and Rational lead to major success in MIS," Document CS-3, Santa Clara, California.

_____. (April 1992). "Rational Environment," Document D-76, Santa Clara, California.

_____. (April 1992). "Rational design support tools," Document D-79, Santa Clara, California.

_____. (April 1992). "Rational R1000 software-engineering server," Document D-80, Santa Clara, California.

_____. (June 1992). "Rational Control: total lifecycle control of Ada projects," Document D-81, Santa Clara, California.

_____. (August 1992). "Rational TestMate," Document D-82, Santa Clara, California.

Rational. (July 1987). "Rational target build utility user's manual," Product Number 4000-00375, Santa Clara, California.

_____. (December 1992). "Rational compilation integrator user's manual," Product Number 4000-00500, Santa Clara, California.

_____. (August 1992). "Insight user's manual," Product Number 4000-00676, Santa Clara, California.

_____. (November 1992). "Testmate user's manual," Product Number 4000-00720, Santa Clara, California.

_____. (Winter 1992). "Rational chosen for Canadian air traffic control software," *The Rational Watch* 2(4), 1-2.

Ripken, K. (1988). "Automated support for design and documentation of large Ada systems." *Proceedings of Milcomp '88 Conference*. .

Royce, W. and Brown, D. (March 1991). "Architecting distributed realtime Ada applications: the Software Architect's Lifecycle Environment." *Proceedings of the Ninth Annual National Conference on Ada Technology*. U.S. Army Communications-Electronics Command, Fort Monmouth, New Jersey, 174-180.

Royce, W., Blankenship, P., Rusis, E., and Willis, B. (March 1991). "Universal Network Architecture Services: a portability case study." *Proceedings of the Ninth Annual National Conference on Ada Technology*. U.S. Army Communications-Electronics Command, Fort Monmouth, New Jersey, 181-187.

Schefström, D. (1990). "Projections from a decade of CASE." *Ada: experiences and prospects*. Proceedings of the Ada-Europe International Conference, Cambridge University Press, Cambridge, 125-138.

Shaw, M. (1986). "Education for the future of software engineering," Technical Memo CMU/SEI-86-TM-5, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pensylvania.

Sommerville, I. ed. (1986). *Software engineering environments*. Peter Peregrinus, London.

Sommerville, I. and Morrison, R. (1987). *Software development with Ada*. Addison-Wesley, Wokingham, England.

Standish, T, A. ed. (June 1978). *Proceedings of the Irvine workshop on alternatives for the environment, certification, and control of the DoD common high order language*.

Statistica. (1991). "SIDPERS-3," white paper, Rockville, Maryland.

Stenning, V. (1986). "An introduction to ISTAR." *Software engineering environments*. I. Sommerville, ed., Peter Peregrinus, London, 1-22.

Suydam, B. (January 1991). "NASA space station leads way in software development," *Defense Electronics* 23(1).

Taft, D. K. (22 January 1990). "IBM getting behind Ada in big way," *Government Computer News* 9(2), 1, 89.

Teitelbaum, T. and Reps, T. (September 1981). "The Cornell Program Synthesizer: a syntax-directed programming environment," *Communications of the ACM* 24(9), 563-573.

Ulvila, J. W. and Brown, R. V. (September/October 1982). "Decision analysis comes of age," *Harvard Business Review*, 130-141.

Wasserman, A. I. (1990). "Tool integration in software engineering environments." *Software engineering environments: international workshop on environments*. F. Long, ed., Lecture Notes in Computer Science, Springer-Verlag, Berlin, 137-149.

Weiderman, N., Altman, N., Borger, M., Klein, M., Landherr, S., Smeaton, R., D'Ippolito, R., Kochmar, J., and Sun, A. (1987). "Evaluation of Ada environments," Technical Report CMU/SEI-87-TR-1, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pensylvania.

Weiderman, N. H., Habermann, A. N., Borger, M., and Klein, M. (December 1986). "A methodology for evaluating environments." *Proceedings of the 2nd ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*. Association for Computing Machinery, New York, 199-207.

Wilson, R. (July 1987). "Ada's influence spreads through defense community," *Computer Design* 26(13), 91-99.

Wood, W., Pethia, R., Gold, L. R., and Firth, R. (April 1988). "A guide to the assessment of software development methods," Technical Report CMU/SEI-88-TR-8, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pensylvania.

Wright Research and Development Center. (August 1988). "Ada Compiler Evaluation Capability (ACEC) technical operating report (TOR) user's guide," Technical Report AFWAL-TR-88-1095, Wright Patterson AFB, Ohio.

Zarrella, P., Smith, D., and Morris, E. J. (1991). "Issues in tool acquisition," Technical Report CMU/SEI-91-TR-8, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pensylvania.

Appendix A
List of Acronyms

ACEC	Ada Compiler Evaluation Capability
ACES	Ada Compiler Evaluation System
AEC	Army Environmental Center
AES	Ada Evaluation System
AFATDS	Advanced Field Artillery Tactical Data System
AJPO	Ada Joint Program Office
AMH	Automated message handler
APSE	Ada programming support environment
ASG	Ada structure graph
ATCCS	Army Tactical Command and Control System
CASE	Computer aided software engineering
CEP	Concept evaluation phase
CICS	Customer Information and Control System
CMM	Capability Maturity Model
CMU	Carnegie Mellon University
CMVC	Configuration management and version control
COTS	Commercial off-the-shelf
CPC	Computer program component
CPCI	Computer program configuration item
DBMS	Data base management system
DIANA	Descriptive intermediate attributed notation for Ada
DSE	Design sensitive editor
DSSE	Development software support environment
EAE	Evaluation of Ada Environments
FIU	Field insertion unit
GUI	Graphical user interface
HI	Human Interface
IPSE	Integrated project support environment
ITL	Infromation Technology Laboratory
KAPSE	Kernel Ada programming support environment
LSE	Language sensitive editor
MAPSE	Minimal Ada programming support environment

MECCA	Multi-element component comparison and analysis
MESC	Magnavox Electronic Systems Company
MIS	Management information systems
MUT	Module under test
NFS	Network File System
OOA	Object-oriented analysis
OOD	Object-oriented design
PDL	Program design language
PSE	Programming support environment
RCI	Rational Compilation Integrator
RCS	Revision Control System
RDF	Rational Design Facility
RISC	Reduced instruction set computer
RPC	Remote procedure call
RWI	Rational Windows Interface
RXI	Rational X Interface
SALE	Software Architect's Lifecycle Environment
SEE	Software engineering environment
SEI	Software Engineering Institute
SIDPERS	Standard Installation/Division Personnel System
SLOC	Source lines of code
STANFINS	Standard Financial System
STANFINS-R	Standard Financial System-Redesign
StP	Software Through Pictures
STSC	Software Technology Support Center
UNAS	Universal Network Architecture Services
VADS	Verdix Ada Development System
WES	Waterways Experiment Station
WIS	WWMCCS Information System
WISCUC	WWMCCS Information System Common User Contract
WWMMCS	World Wide Millitary Comand and Control System
X	X Window System

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 074-0188

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	September 1995	Final Report

4. TITLE AND SUBTITLE

A Study of the Rational Environment

5. FUNDING NUMBERS

6. AUTHOR(S)

William A. Ward, Jr.

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Faculty Court West 20
School of Computer and Information Sciences
University of South Alabama
Mobile, AL 36688

8. PERFORMING ORGANIZATION REPORT NUMBER

Technical Report
USA/CIS-94-TR-02

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Information Technology Laboratory
U.S. Army Engineer Waterways Experiment Station
3909 Halls Ferry Road, Vicksburg, MS 39180-6199

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

Technical Report
ITL-95-9

11. SUPPLEMENTARY NOTES

Available from National Technical Information Service, 5285 Port Royal Road, Springfield, VA 22161.

12a. DISTRIBUTION/AVAILABILITY STATEMENT

Approved for public release; distribution is unlimited

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

This report is intended to be a comprehensive survey of publicly available information on the Rational Environment. Its primary purpose is to introduce potential users of the system to its capabilities by describing its current features and summarizing user experiences. As such, it will also be of interest to students and researchers in the area of software development environments. The report begins by presenting the historical context in which the Rational Environment was developed. So that its use may be more clearly understood by those who have never used software development aids of this type, a brief general discussion of software engineering environments (SEEs) is given as background. This is followed by a description of the Environment itself and tools which may be added to enhance it. The experiences of various Rational users is reported, followed the results of some formal evaluations of the product. A final section presents recommendations on how to successfully use the Environment and some conclusions regarding its capabilities.

14. SUBJECT TERMS	15. NUMBER OF PAGES
Ada, software engineering, software engineering environment, Rational Environment	70
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED		